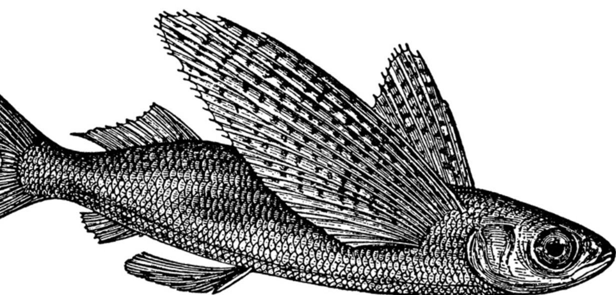
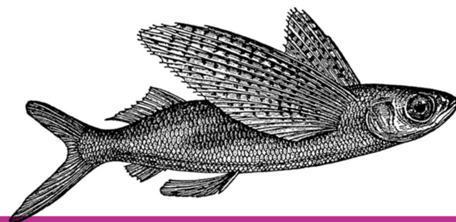
O'REILLY®



图灵程序设计丛书

第2版



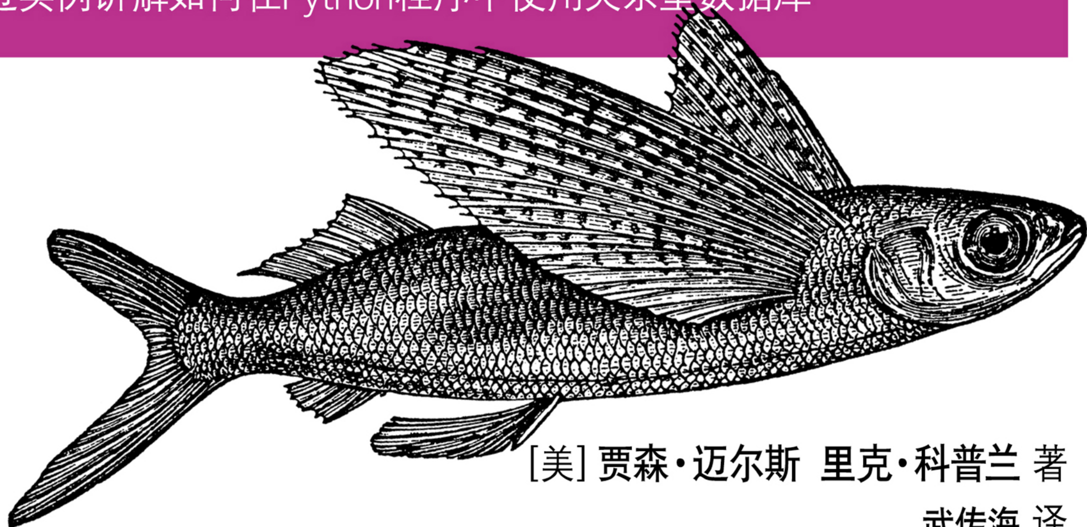


SQLAlchemy

Python数据库实战

Essential SQLAlchemy

通过实例讲解如何在Python程序中使用关系型数据库



[美] 贾森·迈尔斯 里克·科普兰 著

武传海 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

武传海

兴趣广泛，涉猎多个领域，喜欢各类IT技术，掌握丰富的计算机知识，且拥有丰富的翻译经验，尤其擅长翻译IT类技术图书，有多部译著出版，包括《逆向工程核心原理》《利用Python开源工具分析恶意代码》《深入理解MariaDB与MySQL》《Python黑客攻防入门》等，希望能为大家带来更多好译作。

数字版权声明


图灵程序设计丛书

SQLAlchemy：Python数据库实战（第2版）

Essential SQLAlchemy, 2nd Edition

[美] 贾森·迈尔斯　里克·科普兰 著

武传海 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　京

图书在版编目（CIP）数据

SQLAlchemy : Python数据库实战 : 第2版 / (美)
贾森·迈尔斯 (Jason Myers) , (美) 里克·科普兰
(Rick Copeland) 著 ; 武传海译. -- 北京 : 人民邮电
出版社, 2019.8
（图灵程序设计丛书）
ISBN 978-7-115-51630-5

Ⅰ. ①S… Ⅱ. ①贾… ②里… ③武… Ⅲ. ①软件工
具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2019)第140192号

内 容 提 要

本书主要探讨 SQLAlchemy，这个 Python 库在关系型数据库和传统编程之间架起了一座桥梁，有助于 Python 程序员将应用程序连接到关系型数据库。本书首先通过对比的方式介绍了 SQLAlchemy 的两种主要使用模式——SQLAlchemy Core 和 SQLAlchemy ORM，然后探讨了数据库迁移工具 Alembic 的用法，最后快速讲解了 SQLAlchemy 的高级应用。

本书适合 Python 开发人员阅读。

◆ 著　　　　[美] 贾森·迈尔斯 里克·科普兰
　译　　　　武传海
　责任编辑　岳新欣
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800×1000　1/16
　印张：10.75
　字数：242千字　　　　　　2019年8月第1版
　印数：1-3 000册　　　　　2019年8月北京第1次印刷

著作权合同登记号　图字：01-2018-8086号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务还是面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分 SQLAlchemy ORM

第三部分 Alembic

前言

我们周围到处都是数据，数据的存储、更新以及报表制作是每个应用程序应该具备的关键功能。无论你开发的是 Web 应用程序、桌面应用程序还是其他应用程序，你都需要确保可以快速、安全地获取数据。如今，关系型数据库仍然是存放数据最常用的方式之一。

SQL 是一种用来查询和操作数据库数据的强大语言，但有时我们很难将其与所开发的应用程序集成在一起。使用 Python 做开发时，你可能使用字符串创建过在 ODBC 接口上运行的查询，也可能用过 DB API。虽然这些方法可以有效地处理数据，但安全性令人担忧，而且也很难对数据库做出调整。

本书主要探讨 SQLAlchemy，这是一个非常强大又相当灵活的 Python 库，它在关系型数据库和传统编程之间架起了一座桥梁。虽然 SQLAlchemy 允许我们使用原始 SQL 执行查询，但是它更鼓励我们使用其提供的高级方法来查询和更新数据库。SQLAlchemy 提供的方法用起来很友好，而且具有鲜明的 Python 风格。SQLAlchemy 还提供了许多很棒的工具，通过这些工具，你可以轻松地将应用程序中的类和对象同数据库表映射起来，然后“忘掉它”，或者不断回到模型调优性能上。

SQLAlchemy 功能强大并且十分灵活，但也有点令人望而生畏。SQLAlchemy 教程只讲了这个优秀库中的一小部分内容。另外，尽管在线文档非常多，但大都不适合用来学习 SQLAlchemy，而是更适合用作参考资料。本书既是一本 SQLAlchemy 学习指南，也是一本方便的参考手册，每当在工作中碰到问题，你都可以拿来翻一翻，相信你能很快地从中找到答案。

本书重点讲解 SQLAlchemy 1.0 版本，但所讲的大部分内容同样适用于之前的多个版本。对于 0.8 之后的版本，本书示例只需做小幅调整即可运行，并且大部分示例都要求 SQLAlchemy 版本号不低于 0.5。

本书主要分为三个部分：SQLAlchemy Core、SQLAlchemy ORM 和 Alembic。其中，前两

个部分相互对照。我们特意在每个部分使用相同的示例，以方便你比较 SQLAlchemy 的两种主要用法。阅读本书时，你既可以通读 SQLAlchemy Core 和 SQLAlchemy ORM 这两部分内容，也可以只阅读适合你当前需要的那部分。

本书读者

如果你想学习如何在 Python 程序中使用关系型数据库，或者听说过 SQLAlchemy 并且想了解更多信息，那本书正是为你而写的。为了最大限度地利用本书，你的 Python 技能应该达到中等水平，并且对 SQL 数据库有一定的了解。虽然我们努力将本书内容讲得浅显易懂，但如果你刚刚开始学习 Python，建议你先阅读一下 Bill Lubanovic 写的《Python 语言及其应用》[1] 一书，或者观看 Jessica McKellar 制作的"Introduction to Python"教学视频，这些学习资料都很棒。另外，如果你不熟悉 SQL 和数据库，可以先看看 Alan Beaulieu 编写的《SQL 学习指南》一书。事先掌握这些知识对于学习本书内容很有必要。

如何使用书中示例

本书中的大部分示例都需要在 REPL（"读取－求值－输出"循环）环境中运行。你可以在命令提示符下输入 `python` 来使用内置的 Python REPL 环境。这些示例也可以在 IPython notebook 中正常运行。本书有几部分会教你创建和使用文件，而非使用 REPL，比如第 4 章。大部分示例代码和各章的 Python 文件都以 IPython notebook 形式提供给大家。更多有关 IPython 的内容，请去 IPython 官网学习。

阅读前提

本书假定你已经掌握了有关 Python 语法和语义的基础知识，特别是 Python 2.7 及之后的版本。你尤其应该熟悉 Python 中的迭代和对象，本书中会经常用到它们。本书第二部分讲解面向对象编程和 SQLAlchemy ORM。你还应该了解基本的 SQL 语法和关系理论，因为本书假设你已经熟悉如何定义模式和表，以及 `SELECT`、`INSERT`、`UPDATE` 和 `DELETE` 语句的创建。

排版约定

本书使用如下排版约定。

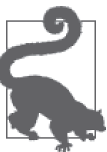
- 黑体字
  表示新术语或重点强调的内容。

注 1：此书已由人民邮电出版社出版，详见 http://www.ituring.com.cn/book/1560。——编者注

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 斜体等宽字体（*`constant width italic`*）
  表示应该替换成用户输入的值，或根据上下文替换的值。



该图标表示提示或建议。



该图标表示一般性说明。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/oreillymedia/essential-sqlalchemy-2e 下载。

本书是要帮你完成工作的。一般来说，如果书中提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用书中的几个代码片段写一个程序无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Essential SQLAlchemy, Second Edition,* by Jason Myers and Rick Copeland (O'Reilly). Copyright 2016 Jason Myers and Rick Copeland, 978-1-4919-1646-9."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O’Reilly Safari



Safari（之前称作 Safari Books Online）一个针对企业、政府、教育者和个人的会员制培训和参考平台。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动式教程和精选播放列表，这些出版商包括 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

要了解更多信息，可以访问 http://www.oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到书的相关信息，包括勘误表[2]、示例代码以及其他信息。本书的网站地址是：http://oreil.ly/1lwEdiw。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

注 2：本书中文版勘误请到 http://www.ituring.com.cn/book/1986 查看和提交。——编者注

致谢

首先，非常感谢 Patrick Altman、Eric Floehr 和 Alex Grönholm 在本书出版前提供的重要反馈。如果没有他们的帮助，本书无疑会有许多技术问题，而且会更难读。

其次，感谢 Mike Bayer，正是在他的推荐下我才动笔撰写本书。感谢 Meghan Blanchette 和 Dawn Schanafelt 督促我完成本书的写作，让我成为一个更好的作者，并且包容了我的种种缺点。还要感谢 Brian Dailey 认真审读了本书部分初稿，并提供了很好的反馈，我们相谈甚欢。

再次，感谢纳什维尔开发社区对我的支持，特别是 Cal Evans、Jacques Woodcock、Luke Stokes、William Golden 这几位朋友。

感谢我的雇主——思科系统公司给了我充足的时间并支持我完成本书。还要感谢 Mechanical Keyboards 公司的 Justin，他为我提供了码字所需的一切。

最后，也是最重要的一点：我要感谢我的妻子，感谢她忍受我大声朗读，放任我去写作，并一直给予我支持和鼓励。我爱你，丹尼丝。

电子书

扫描如下二维码，即可购买本书电子版。



SQLAlchemy入门

SQLAlchemy 库用于与各种数据库交互，你可使用一种类似于 Python 类和语句的方式创建数据模型和查询。SQLAlchemy 库是 Mike Bayer 在 2005 年创建的，现在大大小小很多公司都在使用它。事实上，许多公司都把 SQLAlchemy 看作在 Python 中使用关系型数据库的标准方式。

SQLAlchemy 可用于连接大多数常见的数据库，比如 Postgres、MySQL、SQLite、Oracle 等。SQLAlchemy 还提供了一种为其他关系型数据库添加支持的方式。Amazon Redshift（使用 PostgreSQL 自定义方言）就是 SQLAlchemy 社区添加的数据库支持的一个很好的例子。

在本章中，我们要搞清楚为什么要使用 SQLAlchemy，还要学习它的两种主要模式，以及如何连接到数据库。

为什么使用SQLAlchemy

使用 SQLAlchemy 的首要原因是，它将你的代码从底层数据库及其相关的 SQL 特性中抽象出来。SQLAlchemy 使用功能强大的常见语句和类型，确保其 SQL 语句为每种数据库和供应商正确、高效地构建出来，而无须你考虑这些。同时，这使得将逻辑从 Oracle 迁移到 PostgreSQL 或从应用程序数据库迁移到数据仓库变得很容易。它还有助于确保数据在提交到数据库之前得到“净化”并正确地进行了转义。这可以避免一些常见的问题，比如 SQL 注入攻击。

SQLAlchemy 提供了两种主要的使用模式——SQL 表达式语言（通常称为 Core）和 ORM，这为我们使用 SQLAlchemy 提供了很大的灵活性。这两种模式可以单独使用，也可以一起使用，具体用法取决于你的喜好以及应用程序的需求。

SQLAlchemy Core和SQL表达式语言

SQL 表达式语言允许我们以 Python 方式使用常见的 SQL 语句和表达式，它是对标准 SQL

语言的简单抽象。SQL 表达式语言关注的是实际的数据库模式，它在经过标准化之后为大量后端数据库提供了一种一致性的语言。SQL 表达式语言也是 SQLAlchemy ORM 的基础。

ORM

SQLAlchemy ORM 类似于你在其他语言中遇到的对象关系映射（ORM）。它关注应用程序的领域模型，并利用工作单元模式来维护对象状态。它还在 SQL 表达式语言之上做了进一步的抽象，使用户能够以惯用的方式工作。你可以组合或搭配使用 ORM 与 SQL 表达式语言，从而创建出功能更为强大的应用程序。ORM 中用到了一个声明式系统，该系统类似于许多其他 ORM 中使用的 Active-Record 系统，比如 Ruby on Rails 中使用的那个。

虽然 ORM 极其有用，但你必须记住，类的关联方式和底层数据库关系的工作方式是有区别的。第 6 章会更加全面地探讨这些方式如何影响你的实现。

选择SQLAlchemy Core还是SQLAlchemy ORM

在使用 SQLAlchemy 构建应用程序之前，你要决定主要使用 ORM 还是 Core。选择使用 Core 还是 ORM 作为应用程序的主要数据访问层，通常取决于多个因素和个人偏好。

Core 和 ORM 使用的语法略有不同，但它们之间最大的区别是把数据看作模式还是业务对象。SQLAlchemy Core 的视图以模式为中心，与传统 SQL 一样，它关注的是表、键和索引结构。SQLAlchemy Core 在数据仓库、报表、分析和其他场景中（在这些场景中，严格控制对未建模的数据进行查询或操作将非常有用）可以大放异彩。强大的数据库连接池和结果集优化非常适合处理大量数据，甚至适合处理多个数据库的数据。

但是，如果你主要关注的是领域驱动设计，那么选用 ORM 会更好，它会帮你把元数据和业务对象中的底层模式和结构封装起来。这种封装使得和数据库的交互变得更加容易，就像在使用普通的 Python 代码一样。大多数常见的应用程序都可以通过这种方式进行建模。ORM 还可以有效地把领域驱动设计注入到遗留的应用程序或者到处是原始 SQL 语句的应用程序中。微服务也能从对底层数据库的抽象中获益，它使得开发人员可以只关注正在实现的流程。

但是，由于 ORM 构建在 SQLAlchemy Core 之上，所以你也可以借助 ORM 使用 Oracle 数据仓库和 Amazon Redshift 这样的服务，就像和 MySQL 交互一样。因此，ORM 很适合用来合并业务对象和仓库数据。

下面列出了几种应用场景，了解它们有助于你在 Core 和 ORM 中做出最佳选择。

- 虽然你使用的框架中已经内置了 ORM，但你希望添加更强大的报表功能，请选用 Core。
- 如果你想在一个以模式为中心的视图中查看数据（用法类似于 SQL），请使用 Core。

- 如果你的数据不需要业务对象，请使用 Core。
- 如果你要把数据看作业务对象，请使用 ORM。
- 如果你想快速创建原型，请使用 ORM。
- 如果你需要同时使用业务对象和其他与问题域无关的数据，请组合使用 Core 和 ORM。

你已经了解了 SQLAlchemy 的结构以及 Core 和 ORM 之间的区别，接下来开始安装并使用 SQLAlchemy 来连接数据库。

安装SQLAlchemy并连接到数据库

SQLAlchemy 可以与 Python 2.6、Python 3.3 和 PyPy 2.1 或更高版本一起使用。建议使用 pip 来安装 SQLAlchemy（使用命令 `pip install sqlalchemy`）。值得注意的是，还可以使用 easy_install 和 distutils 来安装它。不过，相比之下，使用 pip 安装更简单。安装过程中，SQLAlchemy 会尝试构建一些 C 扩展，这些扩展可以加快结果集的处理速度，同时提高内存使用效率。如果你的系统中缺少编译器，所以你想禁用这些扩展，那么可以使用 `--global-option=--without-cextensions`。请注意，如果没有 C 扩展，SQLAlchemy 的性能会受到影响。你应该在带有 C 扩展的系统上测试代码，然后再进行优化。

安装数据库驱动程序

默认情况下，SQLAlchemy 直接支持 SQLite3，不需要额外安装驱动程序。不过，在连接其他数据库时需要额外安装一个数据库驱动程序，并且该驱动程序要遵守标准的 Python DBAPI 规范（PEP-249）。这些 DBAPI 为各个数据库服务器所用的方言提供了基础，并且为不同数据库服务器和版本中的独特特性提供了支持。虽然许多数据库都有多种 DBAPI 可用，但我们将只介绍那些最常用的。

- PostgreSQL
  Psycopg2 为 PostgreSQL 的各个版本和特性提供了广泛的支持，你可以使用 `pip install psycopg2` 命令安装它。

- MySQL
  PyMySQL 是我用来连接 MySQL 数据库服务器的首选 Python 库。可以使用 `pip install pymysql` 命令安装它。SQLAlchemy 支持 MySQL 4.1 及更高版本，这是由 MySQL 的密码工作方式造成的。另外，如果某些语句只在 MySQL 的某个版本中可用，那么，对于那些不支持这些语句的 MySQL 版本，SQLAlchemy 不会为它们提供使用这些语句的方法。如果 SQLAlchemy 中的某个组件或函数在你的环境下不起作用，那你首先要做的是查看一下 MySQL 文档。

- 其他

  SQLAlchemy 还可以与 Drizzle、Firebird、Oracle、Sybase 和 Microsoft SQL Server 一起使用。SQLAlchemy 社区还为许多其他的数据库提供了外部方言，如 IBM DB2、Informix、Amazon Redshift、EXASolution、SAP SQL Anywhere、Monet 等。此外，SQLAlchemy 还对创建方言提供了很好的支持，第 7 章将讲解相关内容。

既然我们已经安装好 SQLAlchemy 和 DBAPI 了，接下来创建一个引擎去连接数据库。

连接到数据库

要连接到数据库，需要先创建一个 SQLAlchemy 引擎。SQLAlchemy 引擎为数据库创建一个公共接口来执行 SQL 语句。这是通过包装数据库连接池和方言来实现的，这样它们就可以一起工作，提供对后端数据库的统一访问。如此一来，我们的 Python 代码就不必考虑数据库之间或 DBAPI 之间的差异了。

SQLAlchemy 提供了一个函数来创建引擎。在这个函数中，你可以指定**连接字符串**，以及其他一些可选的关键字参数。连接字符串是一种特殊格式的字符串，包含如下信息：

- 数据库类型（Postgres、MySQL 等）
- 各数据库类型默认之外的方言（Psycopg2、PyMySQL 等）
- 可选的认证细节（用户名和密码）
- 数据库位置（数据库服务器的文件或主机名）
- 数据库服务器端口（可选）
- 数据库名（可选）

SQLite 数据库连接字符串指定了一个特定的数据库文件或存储位置。示例 P-1 定义了一个存储在当前目录下、名为 cookies.db 的 SQLite 数据库文件，当前目录由第二行代码中的相对路径指定，第三行代码是一个内存数据库，第四行（Unix）和第五行（Windows）中指定了数据库文件的完整路径。在 Windows 平台下，连接字符串如最后一行代码所示；除非你使用原始字符串（`r''`），否则就要使用 `\\` 进行字符串转义。

示例 P-1　为 SQLite 数据库创建引擎

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///cookies.db')
engine2 = create_engine('sqlite:///:memory:')
engine3 = create_engine('sqlite:////home/cookiemonster/cookies.db')
engine4 = create_engine('sqlite:///c:\\Users\\cookiemonster\\cookies.db')
```



`create_engine` 函数会返回一个引擎的实例。但是，在调用需要使用连接的操作（比如查询）之前，它实际上并不会打开连接。

接下来，让我们为名为 mydb 的本地 PostgreSQL 数据库创建一个引擎。为此，我们先从 sqlalchemy 包导入 create_engine 函数。然后，使用 create_engine 函数创建引擎实例。在示例 P-2 中，你会看到我把 postgresql+psycopg2 用作连接字符串的引擎和方言组件，当然只使用 postgres 也可以。相比于隐式方式，我更喜欢显式方式，这也正是“Python 之禅”（Zen of Python）所提倡的。

示例 P-2　为本地 PostgreSQL 数据库创建引擎

```
from sqlalchemy import create_engine
engine = create_engine('postgresql+psycopg2://username:password@localhost:' \
                       '5432/mydb')
```

下面看看位于远程服务器上的 MySQL 数据库。在示例 P-3 中你会看到，在连接字符串之后，我们使用了一个关键字参数 pool_recycle，用来指定回收连接的频率。

示例 P-3　为远程 MySQL 数据库创建引擎

```
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://cookiemonster:chocolatechip'
                       '@mysql01.monster.internal/cookies', pool_recycle=3600)
```



默认情况下，超过 8 小时，MySQL 才会关闭空闲连接。为了解决这个问题，可在创建引擎时把 pool_recycle 设置为 3600，如示例 P-3 所示。

create_engine 函数还有一些可选参数，如下所示。

- echo

  开启这个参数会记录引擎处理的操作，例如 SQL 语句及其参数。默认值为 false。

- encoding

  这个参数用于指定 SQLAlchemy 所使用的字符串编码方式，默认值为 utf-8。大多数 DBAPI 默认支持这种编码。但这个参数并不用来指定后端数据库所使用的编码类型。

- isolation_level

  这个参数用来为 SQLAlchemy 指定隔离级别。例如，使用 Psycopg2 的 PostgreSQL 拥有 READ COMMITTED、READ UNCOMMITTED、REPEATABLE READ、SERIALIZABLE 和 AUTOCOMMIT 选项，默认值为 READCOMMITTED。PyMySQL 也有一样的选项，就 InnoDB 数据库来说，默认值为 REPEATABLEREAD。



可以使用 isolation_level 这个关键字参数为任何给定的 DBAPI 设置隔离级别。此外，还可以通过方言连接字符串中的键 – 值对来设置隔离级别，比如支持这个方法的 Psycopg2。

- `pool_recycle`

  设置这个参数会定期回收数据库连接或者让数据库连接超时。因为前面提到过的连接超时，这对 MySQL 非常重要。该参数的默认值为 -1，表示不超时。

初始化引擎之后，就可以实际打开数据库连接了。这可以通过调用 `connect()` 方法实现，代码如下：

```
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://cookiemonster:chocolatechip' \
                       '@mysql01.monster.internal/cookies', pool_recycle=3600)
connection = engine.connect()
```

现在我们有了一个数据库连接，接下来就可以开始使用 SQLAlchemy Core 或 SQLAlchemy ORM 了。在第一部分中，我们将讲解 SQLAlchemy Core，并学习如何定义和查询数据库。

第一部分

SQLAlchemy Core

既然我们已经可以连接到数据库了，接下来学习一下如何使用 SQLAlchemy Core 为应用程序提供数据库服务。SQLAlchemy Core 允许我们以 Python 方式使用 SQL 命令和数据结构，即所谓的 SQL 表达式语言。SQLAlchemy Core 既可以与 Django 或 SQLAlchemy ORM 一起使用，也可以作为独立的解决方案使用。

第 1 章

模式和类型

我们要做的第一件事是定义数据表存储什么样的数据、数据之间如何相互关联，以及数据上的约束。

为了访问底层数据库，SQLAlchemy 需要用某种东西来代表数据库中的数据表。为此，可以使用下面三种方法中的一种：

- 使用用户定义的 `Table` 对象
- 使用代表数据表的声明式类
- 从数据库中推断

本章重点讲解第一种方法，因为它正是 SQLAlchemy Core 所采用的方式。在掌握了基础知识之后，我们会在后面章节中学习另外两种方法。`Table` 对象包含一系列带有类型的列和属性，它们与一个常见的元数据容器相关联。本章我们会学习模式定义，首先来看看 SQLAlchemy 中有哪些类型可用来构建表格。

1.1 类型

在 SQLAlchemy 中有四种类型可用：

- 通用类型
- SQL 标准类型
- 厂商自定义类型
- 用户定义类型

SQLAlchemy 定义了大量通用类型，它们是从各后端数据库所支持的实际 SQL 类型中抽象出来的。这些类型在 `sqlalchemy.types` 模块中都可用，为方便起见，它们在 `sqlalchemy` 模块中也可用。接下来看看这些通用类型为什么有用。

Boolean 通用类型一般使用 `BOOLEAN` SQL 类型，在 Python 端处理成 true 或 false。但是，它在不支持 `BOOLEAN` 类型的后端数据库上使用 `SMALLINT`。幸运的是 SQLAlchemy 隐藏了这个小细节，你可以坚信，你创建的所有查询或语句都能针对该类型的字段正确操作，而不管你使用的是哪种类型的数据库。你只需要在 Python 代码中处理 true 或 false 即可。在数据库迁移或分割后端系统（其中数据仓库是一种数据库类型，事务性系统是另外一种数据库类型）时，这种行为使得通用类型非常强大、有用。通用类型在 Python 和 SQL 中分别对应的类型如表 1-1 所示。

表1-1：通用类型及对应关系

SQLAlchemy	Python	SQL
`BigInteger`	`int`	`BIGINT`
`Boolean`	`bool`	`BOOLEAN` 或 `SMALLINT`
`Date`	`datetime.date`	`DATE(SQLite:STRING)`
`DateTime`	`datetime.datetime`	`DATETIME (SQLite: STRING)`
`Enum`	`str`	`ENUM` 或 `VARCHAR`
`Float`	`float` 或 `Decimal`	`FLOAT` 或 `REAL`
`Integer`	`int`	`INTEGER`
`Interval`	`datetime.timedelta`	`INTERVAL` 或 `DATE`（从纪元开始）
`LargeBinary`	`byte`	`BLOB` 或 `BYTEA`
`Numeric`	`decimal.Decimal`	`NUMERIC` 或 `DECIMAL`
`Unicode`	`unicode`	`UNICODE` 或 `VARCHAR`
`Text`	`str`	`CLOB` 或 `TEXT`
`Time`	`datetime.time`	`DATETIME`



学习这些通用类型很重要，因为你会经常使用和定义它们。

除了表 1-1 中列出的通用类型外，你还可以使用 SQL 标准类型和厂商自定义类型。当通用类型因其类型或现有模式中指定的特定类型而无法在数据库模式中按照需要使用时，我们通常会使用这两种类型。`CHAR` 和 `NVARCHAR` 类型就是很好的例子，它们都要求使用正确的 SQL 类型而不仅仅是通用类型。如果我们使用的是在使用 SQLAlchemy 之前就已经定义好的数据库模式，那么就要准确地匹配类型。请务必记住，SQL 标准类型的行为和可用性因数据库而异。SQL 标准类型在 `sqlalchemy.types` 模块中是可用的。为了将 SQL 标准类型和通用类型区分开，标准类型全部采用大写字母。

厂商自定义类型和 SQL 标准类型一样有用，但是它们只适用于特定的后端数据库。可以通过所选方言的文档或 SQLALchemy 站点确定有哪些类型可用。它们在 `sqlalchemy.dialects` 模块中都是可用的，并且每种数据库方言都有若干子模块。同样，这些类型采用的全是大写字母，以便同通用类型区分开。有时，我们可能想使用 PostgreSQL 强大的 JSON 字段，为此可以使用下面的语句来实现：

```
from sqlalchemy.dialects.postgresql import JSON
```

现在，我们可以定义 JSON 字段了。稍后，在我们的应用程序中，PostgreSQL 专用的许多 JSON 函数都会用到它，比如 `array_to_json` 函数。

你还可以自定义类型，以便用你选择的方式存储数据。例如，在把字符放入数据库记录时，将其放在存储在 `VARCHAR` 列中的文本的前面，并在从记录中获取这个字段时去掉它们。在处理现有系统仍然使用的遗留数据时，这可能很有用，因为这种类型的前缀在新应用程序中可能没用或者并不重要。

我们已经学习了用于创建数据表的四种变体类型，接下来看看数据库结构是如何通过元数据结合在一起的。

1.2　元数据

元数据用来把数据库结构结合在一起，以便在 SQLAlchemy 中快速访问它。一般可以把元数据看作一种 `Table` 对象目录，其中包含与引擎和连接有关的信息。这些表可以通过字典 `MetaData.tables` 来访问。读操作是线程安全的，但是表的创建并不是完全线程安全的。在把对象绑定到元数据之前，需要先导入并初始化元数据。接下来，初始化 `MetaData` 对象的一个实例，在本章的其余示例中我们会用它来保存信息目录：

```
from sqlalchemy import MetaData
metadata = MetaData()
```

到这里，我们已经有了用来保存数据库结构的方法，接下来开始定义表。

1.3　表

在 SQLAlchemy Core 中，我们通过调用 `Table` 构造函数来初始化 `Table` 对象。我们要在构造函数中提供 `MetaData` 对象（元数据）和表名，任何其他参数都被认为是列对象。此外，还有一些额外的关键字参数，它们用来开启相关特性，相关内容稍后讲解。列对象对应于数据表中的各个字段。列是通过调用 `Column()` 函数创建的，我们需要为这个函数提供列名、类型，以及其他表示 SQL 结构和约束的参数。在本章的其余部分中，我们将创建一组表，并在第一部分中使用。在示例 1-1 中，我们创建了一个表，用于为线上 cookie 配送服务存储 cookie 库存。

示例 1-1　实例化 Table 对象和列

```
from sqlalchemy import Table, Column, Integer, Numeric, String, ForeignKey

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True), ❶
    Column('cookie_name', String(50), index=True), ❷
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2)) ❸
)
```

❶ 请注意我们把此列标记为表的主键的方式。稍后会详细介绍。

❷ 创建 cookie 名称索引，以加快该列的查询速度。

❸ 这个列包含多个参数，有长度和精度，比如 11.2，其中长度最多为 11 位数字，精度为两位小数。

在深入了解表之前，需要先了解表的基本构造块：列。

1.3.1　列

列用来定义数据表中的字段，它们提供了通过关键字参数定义其他约束的主要方法。不同类型的列的主要参数是不一样的。例如，String 类型的列的主要参数是长度，而带有小数部分的数字有精度和长度。其他大部分类型没有主要参数。



有时你会看到有些 String 列没有长度这个主要参数。这种行为并没有被普遍支持，例如，MySQL 和其他几个数据库后端就不允许这样做。

除此之外，列还有其他一些关键字参数，这些参数有助于进一步塑造它们的行为。可以把列标记为必需，或者强制它们是唯一的。还可以为列设置默认的初始值，并在记录更新时更改列值。一个常见的用例是那些用来指示何时为日志或审计的目的创建或更新记录的字段。下面看一下示例 1-2 中的这些关键字参数。

示例 1-2　另一个带有更多列选项的表

```
from datetime import datetime
from sqlalchemy import DateTime

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True), ❶
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now), ❷
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.now) ❸
)
```

❶ 我们让这个列是必需的（nullable=False），而且值唯一。

❷ 如果未指定日期，就把当前时间设置为列的默认值。

❸ 通过使用 onupdate，使得每次更新记录时，都把当前时间设置给当前列。



你可能注意到了，我们在设置 default 和 onupdate 时使用的是 datetime.now，而没有调用 datetime.now() 函数。如果调用这个函数，它就会把 default 设置为表首次实例化的时间。通过使用 datetime.now，可以得到实例化和更新每个记录的时间。

我们一直使用列关键字参数来定义表结构和约束，但是，你也可以在 Column 对象之外声明它们。当使用一个现有数据库时，这一点非常重要，因为你必须告诉 SQLAlchemy 数据库中的模式、结构和约束。例如，如果数据库中的已有索引与 SQLAlchemy 使用的默认索引命名方案不匹配，那么你必须手动定义该索引。下面两节内容将演示如何做到这一点。



1.5 节和 1.6 节中的所有命令都包含在 Table 构造函数中，或者通过特殊方法被添加到表中。它们将作为独立语句进行持久化或附加到元数据。

1.3.2 键和约束

键和约束用来确保我们的数据在存储到数据库之前满足某些要求。可以在基本的 SQLAlchemy 模块中找到代表键和约束的对象，其中三个常见的对象导入如下：

```
from sqlalchemy import PrimaryKeyConstraint, UniqueConstraint, CheckConstraint
```

最常见的键类型是主键。主键用作数据库表中每个记录的唯一标识符，用于确保不同表中两个相关数据之间的关系正确。就像你在示例 1-1 和示例 1-2 中看到的那样，只需使用 primary_key 这个关键字参数，就可以让一个列成为主键。还可以通过在多个列上将 primary_key 设置为 True 来定义复合主键。本质上，键会被视为一个元组，其中标记为键的列将按照它们在表中定义的顺序出现。主键也可以在表构造函数的列之后定义，如下面的代码片段所示。可以通过添加多个由逗号分隔的列来创建复合键。如果想像示例 1-2 那样显式地定义键，它看起来会像下面这样：

```
PrimaryKeyConstraint('user_id', name='user_pk')
```

另一个常见的约束是唯一性约束，它用来确保给定字段中不存在重复的两个值。例如，就我们的线上 cookie 配送服务来说，我们希望确保每个客户都有一个唯一的用户名来登录我们的系统。我们还可以为列分配唯一约束，就像前面在 username 列中所做的那样。当然，你也可以手动定义约束，如下所示：

```
UniqueConstraint('username', name='uix_username')
```

示例 1-2 中没有检查约束（check constraint）类型。这种类型的约束用于确保列数据和一组由用户定义的标准相匹配。在下面的示例中，我们会确保 unit_cost 永远不会小于 0.00，因为每个 cookie 的制作都要花费一定的成本（从经济学角度来说，就是没有免费的 cookie！）：

```
CheckConstraint('unit_cost >= 0.00', name='unit_cost_positive')
```

除了键和约束之外，我们可能还希望提高某些字段的查找效率。这就是索引的作用。

1.3.3　索引

索引用来加快字段值的查找速度。在示例 1-1 中，我们在 cookie_name 列上创建了一个索引，因为我们知道以后会经常通过它来进行搜索。在索引创建好之后，你会拥有一个名为 ix_cookies_cookie_name 的索引。还可以使用显式构造类型来定义索引。你可以指定多个列，各列之间用逗号分隔。你还可以添加 unique=True 这个关键字参数，指定索引也必须唯一。当显式地创建索引时，它们会在列之后传递给 Table 构造函数。示例 1-1 中的索引也可以显式地创建：

```
from sqlalchemy import Index
Index('ix_cookies_cookie_name', 'cookie_name')
```

我们还可以创建函数索引，函数索引因所使用的后端数据库而略有不同。这允许你为经常需要基于某些不寻常的上下文进行查询的情况创建索引。例如，如果我们想通过 cookie SKU 和名称进行选择，比如 SKU0001 Chocolate Chip，该怎么办？可以定义下面这样的索引来优化查找：

```
Index('ix_test', mytable.c.cookie_sku, mytable.c.cookie_name))
```

接下来该深入研究关系型数据库最重要的部分了：表关系以及如何定义它们。

1.3.4　关联关系和外键约束

现在我们已经有了一个表，其中的列拥有正确的约束和索引。接下来看看如何在表之间创建关系。我们需要一种跟踪订单的方法，里面有表示每种 cookie 和订购量的行项目。为了帮助理解这些表之间的关系，请参阅图 1-1。

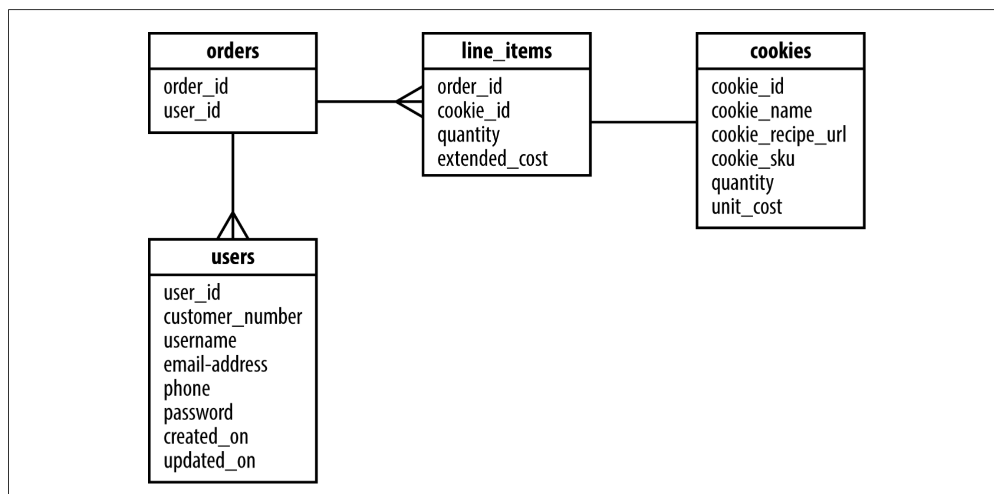


图 1-1：关系可视化

示例 1-3 给出了实现关系的一种方法，即在 `line_items` 表的 `order_id` 列上添加外键，通过 `ForeignKeyConstraint` 来定义两个表之间的关系。示例中，许多行项目都可以出现在单个订单中。但是，如果深入研究 `line_items` 表，你会发现我们还通过 `cookie_id` 这个外键在 `line_items` 表与 `cookies` 表之间建立了关系。这是因为 `line_items` 实际是 `orders` 表和 `cookies` 表之间的一个关联表，其中包含一些额外的数据。关联表用于支持两个其他表之间的多对多关系。表上的单个外键（`ForeignKey`）通常表示一对多的关系，但是，如果一个表上存在多个外键关系，那么它很可能就是关联表。

示例 1-3　定义更多表

```
from sqlalchemy import ForeignKey
orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id')), ❶
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)
```

❶ 请注意，在这个列上，我们使用的是一个字符串，而不是对列的实际引用。

使用字符串而非实际列，这允许我们跨多个模块分离表定义，而且不必担心表的加载顺序。这是因为 SQLAlchemy 只会在第一次访问表名和列时对该字符串执行解析。如果在

ForeignKey 定义中使用了硬引用（hardreference），比如 cookies.c.cookie_id，那它会在模块初始化期间执行解析，并且有可能失败，这取决于表的加载顺序。

你还可以显式地定义一个 ForeignKeyConstraint。外键约束在试图匹配现有数据库模式，以便与 SQLAlchemy 一起使用时会很有用。这与以前创建键、约束和索引以匹配名称模式等的工作方式相同。在表定义中定义外键约束之前，需要先从 sqlalchemy 模块导入 ForeignKeyConstraint。下面的代码演示了如何为 line_items 和 orders 表之间的 order_id 字段创建 ForeignKeyConstraint：

```
ForeignKeyConstraint(['order_id'], ['orders.order_id'])
```

到目前为止，我们一直以 SQLAlchemy 能够理解的方式定义表。如果你的数据库已经存在，并且已经构建好了模式，那接下来你可以着手编写查询了。但是，如果你需要创建完整的模式或添加一个表，就需要知道如何持久化表以实现永久性存储。

1.4　表的持久化

事实上，所有表和模式定义都与 metadata 的实例有关。要想将模式持久化到数据库中，只需调用 metadata 实例的 create_all() 方法，并提供创建表的引擎即可：

```
metadata.create_all(engine)
```

默认情况下，create_all 不会尝试重新创建数据库中已经存在的表，并且它可以安全地运行多次。使用 Alembic 这样的数据库迁移工具来处理对现有表或额外模式所做的更改，要比直接在应用程序代码中进行编码更改更明智（第 11 章会详细地讲解相关内容）。现在我们已经对数据库中的表做了持久化，接下来看看示例 1-4，里面是我们在本章中处理的表的完整代码。

示例 1-4　内存中完整的 SQLite 代码

```
from datetime import datetime

from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, create_engine)
metadata = MetaData()

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2))
)

users = Table('users', metadata,
```

```
    Column('user_id', Integer(), primary_key=True),
    Column('customer_number', Integer(), autoincrement=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.now)
)

orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id'))
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)

engine = create_engine('sqlite:///:memory:')
metadata.create_all(engine)
```

在本章中，我们学习了 SQLAlchemy 如何将元数据用作目录来存储表模式和其他数据，以及如何定义一个拥有多个列和约束的表。我们了解了约束的类型，以及如何在列对象之外显式地构造它们，以匹配现有的模式或命名方案。然后讨论了如何为审计设置默认值和 `onupdate` 值。最后学习了如何把模式持久化或保存到数据库中以供重用。接下来学习如何通过 SQL 表达式语言处理模式中的数据。

第 2 章

使用SQLAlchemy Core处理数据

现在数据库中已经有了表，接下来开始处理这些表中的数据。我们先学习如何插入、检索和删除数据，然后学习如何对数据中的关系进行排序、分组和应用。我们会使用 SQLAlchemy Core 提供的 SQL 表达式语言（SEL）。本章示例将继续使用第 1 章中创建的表。首先学习如何插入数据。

2.1　插入数据

首先，创建一条 insert 语句，用来把我最喜欢的 cookie（chocolate chip）放入 cookies 表中。为此，先调用 cookies 表的 insert() 方法，然后再使用 values() 语句，关键字参数为各个列及相应的值，如示例 2-1 所示。

示例 2-1　插入数据

```
ins = cookies.insert().values(
    cookie_name="chocolate chip",
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",
    cookie_sku="CC01",
    quantity="12",
    unit_cost="0.50"
)
print(str(ins))
```

在示例 2-1 中，print(str(ins)) 语句向我们展示了实际要执行的 SQL 语句：

```
INSERT INTO cookies
    (cookie_name, cookie_recipe_url, cookie_sku, quantity, unit_cost)
VALUES
    (:cookie_name, :cookie_recipe_url, :cookie_sku, :quantity, :unit_cost)
```

在这个 SQL 语句中，我们提供的值被替换为 :column_name，这就是 SQLAlchemy 通过 str() 函数表示参数的方式。参数用来帮助确保数据被正确转义，从而减少 SQL 注入攻击等安全问题。仍然可以通过查看编译后的插入语句来查看参数，因为每个数据库后端处理参数的方式略有不同（这是由方言控制的）。ins 对象的 compile() 方法返回一个 SQLCompiler 对象，该对象允许我们通过 params 属性访问随查询一起发送的实际参数。

```
ins.compile().params
```

这将通过方言编译语句，但不会执行语句，并且我们要访问该语句的 params 属性。

结果如下：

```
{
    'cookie_name': 'chocolate chip',
    'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe.html',
    'cookie_sku': 'CC01',
    'quantity': '12',
    'unit_cost': '0.50'
}
```

现在，我们已经完全了解了插入语句，并且知道了要插入到表中的内容。接下来可以使用 connection 的 execute() 方法把语句发送到数据库，数据库将把记录插入到表中（见示例 2-2）。

示例 2-2　执行插入语句

```
result = connection.execute(ins)
```

还可以通过访问 inserted_primary_key 属性获得刚才插入的记录的 ID：

```
result.inserted_primary_key
[1]
```

让我们快速了解一下调用 execute() 方法时会发生什么。当构建 SQL 表达式语言语句时，比如我们一直在用的插入语句，实际创建的是一个树状结构，这种结构可以采用自上而下的方式快速遍历。当调用 execute 方法时，它会使用传入的语句和其他参数，通过适当的数据库方言的编译器编译语句。编译器通过遍历树状结构构建一个普通的参数化 SQL 语句，而后该语句被返回给 execute 方法，execute 方法再通过调用该方法的连接把 SQL 语句发送到数据库，然后数据库服务器执行语句并返回操作结果。

insert 除了用作 Table 对象的实例方法外，还可以作为独立函数，在你想要逐步构建语句（每次一步）或者在表最初未知时使用。例如，我们公司可能有两个独立的部门，每个部门都有自己单独的库存表。使用示例 2-3 中的 insert 函数，我们就可以使用一条语句替换表。

示例 2-3　插入函数

```
from sqlalchemy import insert
ins = insert(cookies).values( ❶
    cookie_name="chocolate chip",
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",
    cookie_sku="CC01",
    quantity="12",
    unit_cost="0.50"
)
```

❶ 请注意，cookies 表现在用作 insert 函数的参数。



不管是作为 Table 对象的方法，还是作为一种更具生成性的独立函数，insert 都工作得很好，但我更喜欢后一种用法，因为它更接近于人们常见的 SQL 语句。

连接对象的 execute 方法不仅可以接受语句，还可以在语句之后接受关键字参数值。在编译语句时，它会向列列表添加每个关键字参数键，并将它们的每个值添加到 SQL 语句的 VALUES 部分（见示例 2-4）。

示例 2-4　执行语句中的值

```
ins = cookies.insert()
result = connection.execute(
    ins, ❶
    cookie_name='dark chocolate chip', ❷
    cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
    cookie_sku='CC02',
    quantity='1',
    unit_cost='0.75'
)
result.inserted_primary_key
```

❶ 插入语句是 execute 函数的第一个参数，这和前面一样。

❷ 我们以关键字参数的形式把值添加到 execute 函数。

结果如下：

```
[2]
```

虽然这在实际工作中并不常用，但它确实很好地说明了语句在发送到数据库服务器之前是如何编译和组装的。可以通过使用一个字典列表一次插入多条记录，字典里面包含我们要提交的数据。下面使用这种方法把两种 cookie 插入到 cookies 表中（见示例 2-5）。

示例 2-5　插入多条记录

```
inventory_list = [ ❶
    {
        'cookie_name': 'peanut butter',
```

```
        'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html',
        'cookie_sku': 'PB01',
        'quantity': '24',
        'unit_cost': '0.25'
    },
    {
        'cookie_name': 'oatmeal raisin',
        'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html',
        'cookie_sku': 'EWW01',
        'quantity': '100',
        'unit_cost': '1.00'
    }
]
result = connection.execute(ins, inventory_list) ❷
```

❶ 创建 cookie 列表。

❷ 把列表作为 execute 函数的第二个参数。



列表中的字典必须拥有完全相同的键。SQLAlchemy 会根据列表中的第一个字典编译语句，如果后续字典不同，则该语句会失败，因为该语句已经与前面的列一起构建了。

现在 cookies 表中已经有了一些数据，接下来学习如何查询表并获取其中的数据。

2.2 查询数据

构建查询时，要用到 select 函数，它类似于标准 SQL SELECT 语句。首先从 cookies 表获取所有记录（见示例 2-6）。

示例 2-6 简单的 select 函数

```
from sqlalchemy.sql import select
s = select([cookies]) ❶
rp = connection.execute(s)
results = rp.fetchall() ❷
```

❶ 请记住，可以使用 str(s) 查看数据库看到的 SQL 语句，本例中是 SELECT cookies.cookie_id, cookies.cookie_name, cookies.cookie_recipe_url, cookies.cookie_sku, cookies.quantity, cookies.unit_cost FROM cookies。

❷ 这让 rp（ResultProxy）返回所有行。

results 变量现在包含一个列表，里面有 cookies 表中的所有记录：

```
[(1, u'chocolate chip', u'http://some.aweso.me/cookie/recipe.html', u'CC01',
  12, Decimal('0.50')),
 (2, u'dark chocolate chip', u'http://some.aweso.me/cookie/recipe_dark.html',
  u'CC02', 1, Decimal('0.75')),
```

```
 (3, u'peanut butter', u'http://some.aweso.me/cookie/peanut.html', u'PB01',
  24, Decimal('0.25')),
 (4, u'oatmeal raisin', u'http://some.okay.me/cookie/raisin.html', u'EWW01',
  100, Decimal('1.00'))]
```

在前面的示例中，我传递了一个包含 `cookies` 表的列表。`select` 方法需要一个列列表来进行选择，但为了方便起见，它还可以接受 `Table` 对象，并选择表中的所有列。也可以使用 `Table` 对象的 `select` 方法来实现这一点，如示例 2-7 所示。但是，我更喜欢示例 2-6 的写法。

示例 2-7　简单的 select() 方法

```
from sqlalchemy.sql import select
s = cookies.select()
rp = connection.execute(s)
results = rp.fetchall()
```

在深入学习查询之前，需要进一步了解一下 `ResultProxy` 对象。

2.2.1　ResultProxy

`ResultProxy` 是 DBAPI 游标对象的包装器，其主要目标是让语句返回的结果更容易使用和操作。比如，`ResultProxy` 允许使用索引、名称或 `Column` 对象进行访问，从而简化了对查询结果的处理。示例 2-8 演示了这三种方法。能够熟练地使用这三个方法来获取所需的列数据非常重要。

示例 2-8　使用 ResultProxy 处理行

```
first_row = results[0] ❶
first_row[1] ❷
first_row.cookie_name ❸
first_row[cookies.c.cookie_name] ❹
```

❶ 获取 `ResultProxy` 的第一行。

❷ 通过索引访问列。

❸ 通过名称访问列。

❹ 通过 `Column` 对象访问列。

上面三条语句的执行结果都是 `'chocolate chip'`，它们引用的是 `results` 变量的第一条记录中完全相同的数据元素。这种访问灵活性只是 `ResultProxy` 强大功能的一部分。还可以把 `ResultProxy` 用作可迭代对象，对返回的每条记录执行一个操作，并且无须创建另一个变量来保存结果。例如，我们可能希望打印数据库中每种 cookie 的名称（见示例 2-9）。

示例 2-9　迭代 ResultProxy

```
rp = connection.execute(s) ❶
for record in rp:
    print(record.cookie_name)
```

❶ 这里重用了前面的 select 语句。

返回结果如下：

```
chocolate chip
dark chocolate chip
peanut butter
oatmeal raisin
```

除了把 ResultProxy 用作可迭代对象和调用 fetchall() 方法之外，还有其他许多通过 ResultProxy 访问数据的方法。事实上，2.1 节中的所有 result 变量都是 ResultProxys。我们在该节中使用的 rowcount() 和 inserted_primary_key() 方法只是从 ResultProxy 获取信息的另外两种方法。还可以使用如下方法来获取结果。

first()

若有记录，则返回第一个记录并关闭连接。

fetchone()

返回一行，并保持光标为打开状态，以便你做更多获取调用。

scalar()

如果查询结果是包含一个列的单条记录，则返回单个值。

如果想查看结果集中的多个列，可以使用 keys() 方法来获得列名列表。在本章的其余部分中，我们将使用 first、scalar、fetchone 和 fetchall 方法以及 ResultProxy 可迭代对象。

最佳实践

在编写生产代码时，应该遵循如下指导方针。

- 获取单条记录时，要多用 first 方法，尽量不要用 fetchone 和 scalar 方法，因为对程序员来说，first 方法更加清晰。
- 尽量使用可迭代对象 ResultProxy，而不要用 fetchall 和 fetchone 方法，因为前者的内存效率更高，而且我们往往一次只对一条记录进行操作。
- 避免使用 fetchone 方法，因为如果不小心，它会一直让连接处于打开状态。
- 谨慎使用 scalar 方法，因为如果查询返回多行多列，就会引发错误，多行多行在测试过程中经常会丢失。

在前面的示例中，我们每次查询数据库时都会得到各条记录的所有列。通常我们只需要使用这些列中的一部分。如果这些额外列中的数据量很大，就会导致应用程序运行速度变慢，消耗的内存远超预期。尽管 SQLAlchemy 不会向查询或 ResultProxys 添加大量开销，但是，如果查询占用了太多内存，首先要考虑从查询返回的数据是否有问题。接下来看看如何对查询返回的列数进行限制。

2.2.2　控制查询中的列数

为了限制查询返回的列数，需要以列表的形式把要查询的列传递给 `select()` 方法。例如，你想运行一个查询，该查询只返回 cookie 的名称和数量，如示例 2-10 所示。

示例 2-10　只包含 `cookie_name` 和 `quantity`

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
rp = connection.execute(s)
print(rp.keys()) ❶
result = rp.first() ❷
```

❶ 返回所选列的列表，在本例中是 `['cookie_name','quantity']`（这只用来演示，不是必需的）。

❷ 请注意，这条语句只返回第一个结果。

结果如下：

```
(u'chocolate chip', 12),
```

我们已经可以构建简单的 `select` 语句了，接下来看看还可以做些什么来改变 `select` 语句返回结果的方式。首先学习如何改变返回结果的顺序。

2.2.3　排序

如果查看示例 2-10 的所有结果，而不仅仅是第一条记录，你会发现数据并不是按照特定顺序排列的。如果希望返回的列表有特定的顺序，可以使用 `order_by()` 语句，如示例 2-11 所示。示例中，我们希望结果按照现有 cookie 的数量进行排序。

示例 2-11　按 `quantity` 进行升序排列

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
s = s.order_by(cookies.c.quantity)
rp = connection.execute(s)
for cookie in rp:
    print('{} - {}'.format(cookie.quantity, cookie.cookie_name))
```

结果如下：

```
1 - dark chocolate chip
12 - chocolate chip
24 - peanut butter
100 - oatmeal raisin
```

我们先把 `select` 语句保存到变量 `s` 中，而后向变量 `s` 中添加 `order_by` 语句，再将其重新赋给 `s` 变量。这个示例演示了如何以生成式或分步方式编写语句。也可以把 `select` 和 `order_by` 语句放在一行中，如下所示：

```
s = select([...]).order_by(...)
```

然而，当 select 中有完整的列列表，order_by 语句中有 order 列时，它超过了 Python 每行 79 个字符的限制（这是在 PEP8 中建立的）。通过使用生成类型语句，可以保持不超出该限制。在本书中，我们将看到一些例子，其中生成类型语句可以带来额外的好处，例如有条件地向语句中添加内容。现在，尝试按照每行 79 个字符的限制分割语句，这将有助于提升代码的可读性。

如果你想按倒序或降序排列，可使用 desc() 语句。desc() 函数的作用是按降序包装你要排序的特定列，如示例 2-12 所示。

示例 2-12　按照 quantity 进行降序排列

```
from sqlalchemy import desc
s = select([cookies.c.cookie_name, cookies.c.quantity])
s = s.order_by(desc(cookies.c.quantity)) ❶
```

❶ 请注意，这一行中，我们使用 desc() 函数对 cookies.c.quantity 列进行了包装。



desc() 函数还可以作为 Column 对象（比如 cookie.c.quantity.desc()）的方法进行调用。但是，如果在长语句中这样使用，可能会造成阅读困难，所以我总是把 desc() 用作函数。

如果应用程序只需要返回结果的一部分，可以对返回的结果数量进行限制。

2.2.4　限制返回结果集的条数

在前面的示例中，我们使用 first() 或 fetchone() 方法仅获取一行。虽然 ResultProxy 提供了我们请求的那行，但查询实际运行时会访问所有结果，而不仅仅是单个记录。如果想对查询进行限制，可以使用 limit() 函数让 limit 语句成为查询的一部分。比如，你的时间只够制作两批 cookie，你想知道应该制作哪两种 cookie，那么你可以使用前面的有序查询，并添加 limit 语句来返回最需要补充的两种 cookie（见示例 2-13）。

示例 2-13　两种库存量最少的 cookie

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
s = s.order_by(cookies.c.quantity)
s = s.limit(2)
rp = connection.execute(s)
print([result.cookie_name for result in rp]) ❶
```

❶ 这里，我们在列表中使用了 ResultsProxy 的可迭代功能。

结果如下：

```
[u'dark chocolate chip', u'chocolate chip']
```

现在，你已经知道需要烤哪两种 cookie 了，你可能还想知道当前库存中还有多少 cookie。

许多数据库都包含 SQL 函数，这些函数的设计目的是让某些操作可以直接在数据库服务器上使用，比如 SUM。接下来了解一下如何使用 SQL 函数。

2.2.5 内置SQL函数和标签

SQLAlchemy 还可以利用后端数据库中的 SQL 函数。两个非常常用的数据库函数是 `SUM()` 和 `COUNT()`。要使用这两个函数，需要导入 `sqlalchemy.sql.func` 模块。这些函数被包装在它们操作的列上。因此，要获得 cookie 的总数，可以使用示例 2-14。

示例 2-14　计算 cookie 的总数

```
from sqlalchemy.sql import func
s = select([func.sum(cookies.c.quantity)])
rp = connection.execute(s)
print(rp.scalar()) ❶
```

❶ 请注意，这里使用了 `scalar()`，它只返回第一个记录最左边的列。

结果如下：

```
137
```



我总是导入 `func` 模块，因为直接导入 `sum` 可能会引起问题，而且还容易和 Python 内置的 `sum` 函数混淆。

接下来使用 `count` 函数查看一下 `cookies` 表中有几种 cookie（见示例 2-15）。

示例 2-15　统计库存中有几种 cookie

```
s = select([func.count(cookies.c.cookie_name)])
rp = connection.execute(s)
record = rp.first()
print(record.keys()) ❶
print(record.count_1) ❷
```

❶ 显示 `ResultProxy` 中的列。

❷ 自动生成列名，一般格式为：`<func_name>_<position>`。

结果如下：

```
[u'count_1']
4
```

这个列名既烦人又麻烦。另外，如果查询中有多个统计，那我们必须知道它们在语句中出现的次数，并将其合并到列名，因此第四个 `count()` 函数将是 `count_4`。命名应该清

晰、明确，特别是当周围有其他 Python 代码时。不过，值得庆幸的是，SQLAlchemy 提供了 `label()` 函数来解决这个问题。示例 2-16 执行的查询与示例 2-15 一样，但它通过调用 `label()` 函数为我们访问的列起了一个更有用的名字。

示例 2-16　使用 `lable()` 进行重命名

```
s = select([func.count(cookies.c.cookie_name).label('inventory_count')]) ❶
rp = connection.execute(s)
record = rp.first()
print(record.keys())
print(record.inventory_count)
```

❶ 请注意，只在要更改的列对象上调用 `label()` 函数即可。

结果如下：

```
[u'inventory_count']
4
```

我们已经学习了如何限制从数据库返回的列数或行数。接下来学习如何根据指定的条件对查询数据进行过滤。

2.2.6　过滤

对查询进行过滤是通过添加 `where()` 语句来完成的，和在 SQL 中一样。典型的 `where()` 子句包含一个列、一个运算符和一个值或列。可以把多个 `where()` 子句接在一起使用，功能就像传统 SQL 语句中的 AND 一样。在示例 2-17 中，我们将查找名为“chocolate chip”的 cookie。

示例 2-17　使用 `cookie_name` 进行过滤

```
s = select([cookies]).where(cookies.c.cookie_name == 'chocolate chip')
rp = connection.execute(s)
record = rp.first()
print(record.items()) ❶
```

❶ 这里我调用了行对象的 `items()` 方法，该方法返回一个包含列名和列值的列表。

结果如下：

```
[
    (u'cookie_id', 1),
    (u'cookie_name', u'chocolate chip'),
    (u'cookie_recipe_url', u'http://some.aweso.me/cookie/recipe.html'),
    (u'cookie_sku', u'CC01'),
    (u'quantity', 12),
    (u'unit_cost', Decimal('0.50'))
]
```

还可以使用 `where()` 语句来查找所有包含“chocolate”这个词的 cookie 名（见示例 2-18）。

示例 2-18　查找包含 chocolate 的 cookie 名

```
s = select([cookies]).where(cookies.c.cookie_name.like('%chocolate%'))
rp = connection.execute(s)
for record in rp.fetchall():
    print(record.cookie_name)
```

结果如下：

```
chocolate chip
dark chocolate chip
```

在示例 2-18 的 `where()` 语句中，我们将 `cookies.c.cookie_name` 列用作一种 `ClauseElement` 来过滤结果。我们应该花一些时间了解一下 `ClauseElement` 及其提供的其他功能。

2.2.7　ClauseElement

`ClauseElement` 是在子句中使用的实体，一般是表中的列。不过，与列不同的是，`ClauseElement` 拥有许多额外的功能。在示例 2-18 中，我们调用了 `ClauseElement` 的 `like()` 方法。此外还有许多其他方法可供选用，如表 2-1 所示。这些方法的结构和标准 SQL 语句类似。你会在本书中找到各种各样的例子。

表2-1：ClauseElement方法

方　法	用　途
`between(cleft, cright)`	查找在 `cleft` 和 `cright` 之间的列
`concat(column_two)`	连接列
`distinct()`	查找列的唯一值
`in_([list])`	查找列在列表中的位置
`is_(None)`	查找列 None 的位置（通常用于检查 Null 和 None）
`contains(`*`string`*`)`	查找包含 *`string`* 的列（区分大小写）
`endswith(`*`string`*`)`	查找以 *`string`* 结尾的列（区分大小写）
`like(`*`string`*`)`	查找与 *`string`* 匹配的列（区分大小写）
`startswith(`*`string`*`)`	查找以 *`string`* 开头的列（区分大小写）
`ilike(`*`string`*`)`	查找与 *`string`* 匹配的列（不区分大小写）



这些方法也存在相反的版本，例如 `notlike` 和 `notin_()`。`not<方法>` 这种命名约定的唯一例外是不带下划线的 `isnot()` 方法。

除了使用表 2-1 中列出的方法外，还可以在 `where` 子句中使用运算符。大多数运算符的工作方式和你预想的一样。接下来详细地讲解一下运算符，它们之间存在一些差异。

2.2.8 运算符

到目前为止，我们用来过滤数据的方法有两种：一是利用列是否等于某个值，二是使用 ClauseElement 的方法，比如 like()。然而，还可以使用许多常见的运算符来过滤数据。SQLAlchemy 对大多数标准 Python 运算符做了重载，包括所有标准的比较运算符（==、!=、<、>、<=、>=），它们的功能和在 Python 语句中完全一样。在与 None 比较时，== 运算符被重载为 IS NULL 语句。算术运算符（\+、-、*、/ 和 %）还可以用来对独立于数据库的字符串做连接处理，如示例 2-19 所示。

示例 2-19　使用 \+ 连接字符串

```
s = select([cookies.c.cookie_name, 'SKU-' + cookies.c.cookie_sku])
for row in connection.execute(s):
    print(row)
```

结果如下：

```
(u'chocolate chip', u'SKU-CC01')
(u'dark chocolate chip', u'SKU-CC02')
(u'peanut butter', u'SKU-PB01')
(u'oatmeal raisin', u'SKU-EWW01')
```

运算符的另一个常见用法是根据多个列来计算值。在处理财务数据或统计数据的应用程序和报表中，你经常要这样做。示例 2-20 是计算库存价值的一个例子。

示例 2-20　计算各种 cookie 的库存价值

```
from sqlalchemy import cast ❶
s = select([cookies.c.cookie_name,
          cast((cookies.c.quantity * cookies.c.unit_cost),
               Numeric(12,2)).label('inv_cost')]) ❷
for row in connection.execute(s):
    print('{} - {}'.format(row.cookie_name, row.inv_cost))
```

❶ Cast() 是另一个允许做类型转换的函数。本例中，我们要获取 6.0000000000 这样的结果，因此需要进行强制类型转换，使其看起来像货币。在 Python 中，还可以使用 print('{} - {:.2f}'.format(row.cookie_name, row.inv_cost)) 完成相同的任务。

❷ 请注意，这里再次使用 label() 函数对列进行重命名。如果不进行重命名，列就会被命名为 anon_1，因为操作本身不会产生名字。

结果如下：

```
chocolate chip - 6.00
dark chocolate chip - 0.75
peanut butter - 6.00
oatmeal raisin - 100.00
```

2.2.9　布尔运算符

SQLAlchemy 还支持 SQL 布尔运算符 AND、OR 和 NOT，它们用位运算符（&、| 和 ~）来表示。受 Python 运算符优先级规则的影响，在使用 AND、OR 和 NOT 重载运算符时一定要特别小心。比如，& 比 < 优先级高，所以当你写 A < B & C < D 时，你想得到的是 (A < B) &(C < D)，而实际得到的却是 A < (B&C) < D。请尽量使用连接词（conjunction），不要用这些重载运算符，因为连接词会让你的代码更清晰易懂。

通常，我们希望用包含和排除的方式把多个 where() 子句链接在一起，这可以通过使用连接词来完成。

2.2.10　连接词

为了实现某种期望的效果，我们既可以把多个 where() 子句链接在一起，也可以使用连接词来实现，而且使用连接词的可读性更好、功能性更强。SQLAlchemy 中的连接词是 and_()、or_() 和 not_()。如果想获得价格低于某个数且数量超过指定值的 cookie 列表，可以使用示例 2-21 中的代码。

示例 2-21　使用 and() 连接词

```
from sqlalchemy import and_, or_, not_
s = select([cookies]).where(
    and_(
        cookies.c.quantity > 23,
        cookies.c.unit_cost < 0.40
    )
)
for row in connection.execute(s):
    print(row.cookie_name)
```

or_() 函数的作用与 and_() 函数正好相反，只要记录满足其中一个条件，就会被选出来。如果想搜索库存中数量在 10 到 50 之间或名字包含 chip 的 cookie，可以使用示例 2-22 中的代码。

示例 2-22　使用 or() 连接词

```
from sqlalchemy import and_, or_, not_
s = select([cookies]).where(
    or_(
        cookies.c.quantity.between(10, 50),
        cookies.c.cookie_name.contains('chip')
    )
)
for row in connection.execute(s):
    print(row.cookie_name)
```

结果如下：

```
chocolate chip
dark chocolate chip
peanut butter
```

not_() 函数的工作方式与其他连接词类似，用来选择那些与指定条件不匹配的记录。到这里，我们已经可以轻松地查询数据了，接下来该学习如何更新现有数据了。

2.3 更新数据

update() 方法和前面用过的 insert() 方法很相似，它们的语法几乎完全一样，但是 update() 可以指定一个 where 子句，用来指出要更新哪些行。与插入语句一样，更新语句可以由 update() 函数或者表格的 update() 方法来创建。如果省略 where 子句，则表示要更新表中的所有行。

例如，假设你已经完成了库存所需的巧克力饼干（chocolate chip cookies）的烘焙工作。在示例 2-23 中，我们先通过更新查询把烘焙好的巧克力饼干添加到当前库存中，再查看当前库存中巧克力饼干的数量。

示例 2-23　更新数据

```
from sqlalchemy import update
u = update(cookies).where(cookies.c.cookie_name == "chocolate chip")
u = u.values(quantity=(cookies.c.quantity + 120)) ❶
result = connection.execute(u)
print(result.rowcount) ❷
s = select([cookies]).where(cookies.c.cookie_name == "chocolate chip")
result = connection.execute(s).first()
for key in result.keys():
    print('{:>20}: {}'.format(key, result[key]))
```

❶ 使用生成方法构建语句。

❷ 打印更新的行数。

结果如下：

```
1
          cookie_id: 1
        cookie_name: chocolate chip
  cookie_recipe_url: http://some.aweso.me/cookie/recipe.html
         cookie_sku: CC01
           quantity: 132
          unit_cost: 0.50
```

除了更新数据之外，某些时候，我们还想从表中删除某些数据，接下来就学习删除数据的方法。

2.4 删除数据

创建删除语句时，既可以使用 delete() 函数，也可以使用表（包含待删数据的表）的 delete() 方法。与 insert() 和 update() 不同，delete() 不接收值参数，只接收一个可选的 where 子句，用来指定删除范围（省略 where 子句表示删除表中的所有行）。请看示例 2-24。

示例 2-24 删除数据

```
from sqlalchemy import delete
u = delete(cookies).where(cookies.c.cookie_name == "dark chocolate chip")
result = connection.execute(u)
print(result.rowcount)

s = select([cookies]).where(cookies.c.cookie_name == "dark chocolate chip")
result = connection.execute(s).fetchall()
print(len(result))
```

结果如下：

```
1
0
```

现在，让我们综合运用前面学过的内容向 users、orders 和 line_items 表中加载一些数据。你可以直接复制下面这些代码进行添加，但最好花点时间尝试使用不同的插入数据的方法。

```
customer_list = [
    {
        'username': 'cookiemon',
        'email_address': 'mon@cookie.com',
        'phone': '111-111-1111',
        'password': 'password'
    },
    {
        'username': 'cakeeater',
        'email_address': 'cakeeater@cake.com',
        'phone': '222-222-2222',
        'password': 'password'
    },
    {
        'username': 'pieguy',
        'email_address': 'guy@pie.com',
        'phone': '333-333-3333',
        'password': 'password'
    }
]
ins = users.insert()
result = connection.execute(ins, customer_list)
```

现在我们有了客户，接下来开始把他们的订单和行项目输入到系统中：

```
ins = insert(orders).values(user_id=1, order_id=1)
result = connection.execute(ins)
ins = insert(line_items)
order_items = [
    {
        'order_id': 1,
        'cookie_id': 1,
        'quantity': 2,
        'extended_cost': 1.00
    },
    {
        'order_id': 1,
        'cookie_id': 3,
        'quantity': 12,
        'extended_cost': 3.00
    }
]
result = connection.execute(ins, order_items)
ins = insert(orders).values(user_id=2, order_id=2)
result = connection.execute(ins)
ins = insert(line_items)
order_items = [
    {
        'order_id': 2,
        'cookie_id': 1,
        'quantity': 24,
        'extended_cost': 12.00
    },
    {
        'order_id': 2,
        'cookie_id': 4,
        'quantity': 6,
        'extended_cost': 6.00
    }
]
result = connection.execute(ins, order_items)
```

第 1 章中，我们学习了如何定义 `ForeignKeys`（外键）和关系，但是到目前为止，我们还没有用它们做过查询。下面就来看看这些关系。

2.5　连接

现在，让我们学习如何使用 `join()` 和 `outerjoin()` 方法来查询相关数据。例如，为了完成 cookiemon 客户所下的订单，我们需要确认每种 cookie 的订购量。总共需要使用 3 个连接才能获取 cookie 的名称。需要注意的是，根据连接在关系中的使用方式，可能需要重新组织语句中的 `from` 部分。为此，可以使用 SQLAlchemy 提供的 `select_from()` 子句。通过使用 `select_from()`，我们可以用指定的 `from` 子句替换 SQLAlchemy 生成的整个 `from` 子句（见示例 2-25）。

示例 2-25　使用连接查询多个表

```
columns = [orders.c.order_id, users.c.username, users.c.phone,
           cookies.c.cookie_name, line_items.c.quantity,
           line_items.c.extended_cost]
cookiemon_orders = select(columns)
cookiemon_orders = cookiemon_orders.select_from(orders.join(users).join( ❶
                        line_items).join(cookies)).where(users.c.username ==
                            'cookiemon')
result = connection.execute(cookiemon_orders).fetchall()
for row in result:
    print(row)
```

❶ 请注意，这里，我们让 SQLAlchemy 把关系连接用作 `from` 子句。

结果如下：

```
(u'1', u'cookiemon', u'111-111-1111', u'chocolate chip', 2, Decimal('1.00'))
(u'1', u'cookiemon', u'111-111-1111', u'peanut butter', 12, Decimal('3.00'))
```

我们得到如下 SQL 语句：

```
SELECT orders.order_id, users.username, users.phone, cookies.cookie_name,
line_items.quantity, line_items.extended_cost FROM users JOIN users ON
users.user_id = orders.user_id JOIN line_items ON orders.order_id =
line_items.order_id JOIN cookies ON cookies.cookie_id = line_items.cookie_id
WHERE users.username = :username_1
```

此外，获取所有用户（包括那些当前没有订单的用户）的订单数量也很有用。为此，必须使用 `outerjoin()` 方法，并且要对连接顺序多加注意，因为应用 `outerjoin()` 方法的表会返回所有结果（见示例 2-26）。

示例 2-26　使用外连接查询多个表

```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders)) ❶
all_orders = all_orders.group_by(users.c.username)
result = connection.execute(all_orders).fetchall()
for row in result:
    print(row)
```

❶ SQLAlchemy 知道如何连接 `users` 和 `orders` 表，因为 `orders` 表中定义了外键。

结果如下：

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieguy', 0)
```

到目前为止，我们一直在查询中使用和连接不同的表。但是，如果我们有一个自引用（self-referential）的表，比如员工和老板的关系表，该怎么办呢？为了便于读取和理解，SQLAlchemy 引入了别名。

2.6 别名

使用连接时，常常需要多次引用一个表。在 SQL 中，这是通过在查询中使用**别名**来实现的。例如，假设我们有下面一个（不完整的）表，它用来描述某个组织中的上下级关系：

```
employee_table = Table(
    'employee', metadata,
    Column('id', Integer, primary_key=True),
    Column('manager', None, ForeignKey('employee.id')),
    Column('name', String(255)))
```

假设现在我们要选择由 Fred 管理的所有员工。在 SQL 中，可以这样写：

```
SELECT employee.name
FROM employee, employee AS manager
WHERE employee.manager_id = manager.id
    AND manager.name = 'Fred'
```

在这种情况下，SQLAlchemy 允许我们使用 `alias()` 函数或方法来实现：

```
>>> manager = employee_table.alias('mgr')
>>> stmt = select([employee_table.c.name],
...               and_(employee_table.c.manager_id==manager.c.id,
...                    manager.c.name=='Fred'))
>>> print(stmt)
SELECT employee.name
FROM employee, employee AS mgr
WHERE employee.manager_id = mgr.id AND mgr.name = ?
```

SQLAlchemy 还可以自动选择别名，这可以避免出现名称冲突：

```
>>> manager = employee_table.alias()
>>> stmt = select([employee_table.c.name],
...               and_(employee_table.c.manager_id==manager.c.id,
...                    manager.c.name=='Fred'))
>>> print(stmt)
SELECT employee.name
FROM employee, employee AS employee_1
WHERE employee.manager_id = employee_1.id AND employee_1.name = ?
```

做数据报表时，对数据进行分组很有用。接下来学习如何对数据进行分组。

2.7 分组

使用分组时，你需要一个或多个列来做分组，以及一个或多个列来做统计，比如计数、求和等，就像在普通 SQL 中所做的那样。下面，让我们按客户获得订单数量（见示例 2-27）。

示例 2-27　数据分组

```
columns = [users.c.username, func.count(orders.c.order_id)] ❶
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username) ❷
result = connection.execute(all_orders).fetchall()
for row in result:
    print(row)
```

❶ 使用 count() 统计订单数。

❷ 根据用户名进行分组。

结果如下：

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieguy', 0)
```

在前面的例子中，构建语句时，我们已经用过生成式构建方法。接下来详细讲讲。

2.8　链式调用

本章中，我们已经多次用过链式调用（chaining），只是没有明说。构建查询时，如果想把逻辑清晰地表达出来，那使用链式调用会特别有用。如果我们想创建一个函数，让其为我们获取订单数据，则可以使用示例 2-28 所示的代码。

示例 2-28　链式调用

```
def get_orders_by_customer(cust_name):
    columns = [orders.c.order_id, users.c.username, users.c.phone,
               cookies.c.cookie_name, line_items.c.quantity,
               line_items.c.extended_cost]
    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(
        users.join(orders).join(line_items).join(cookies))
    cust_orders = cust_orders.where(users.c.username == cust_name)
    result = connection.execute(cust_orders).fetchall()
    return result

get_orders_by_customer('cakeeater')
```

结果如下：

```
[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]
```

但是，如果只想获得那些已经发货或者还没有发货的订单，该怎么办呢？为此，必须编写其他函数来支持额外的过滤选项，或者使用条件来构建查询链。另外，我们可能还需要一个选项，用来指定是否包含细节。利用这种把查询和子句链接在一起的方法，我们可以做

出功能强大的报表，以及构建出复杂的查询（见示例 2-29）。

示例 2-29 带条件的链式调用

```
def get_orders_by_customer(cust_name, shipped=None, details=False):
    columns = [orders.c.order_id, users.c.username, users.c.phone]
    joins = users.join(orders)
    if details:
        columns.extend([cookies.c.cookie_name, line_items.c.quantity,
                        line_items.c.extended_cost])
        joins = joins.join(line_items).join(cookies)
    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins)
    cust_orders = cust_orders.where(users.c.username == cust_name)
    if shipped is not None:
        cust_orders = cust_orders.where(orders.c.shipped == shipped)
    result = connection.execute(cust_orders).fetchall()
    return result

get_orders_by_customer('cakeeater') ❶

get_orders_by_customer('cakeeater', details=True) ❷

get_orders_by_customer('cakeeater', shipped=True) ❸

get_orders_by_customer('cakeeater', shipped=False) ❹

get_orders_by_customer('cakeeater', shipped=False, details=True) ❺
```

❶ 获取所有订单。

❷ 获取所有订单的细节。

❸ 只获取已经发货的订单。

❹ 获取还没发货的订单。

❺ 获取还没发货的订单细节。

结果如下：

```
[(u'2', u'cakeeater', u'222-222-2222')]

[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

[]

[(u'2', u'cakeeater', u'222-222-2222')]

[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]
```

到目前为止，我们在所有示例中使用的都是 SQL 表达式语言。或许，你还想知道 SQLAlchemy 是否也支持标准的 SQL 语句。接下来，让我们了解一下。

2.9 原始查询

事实上，在 SQLAlchemyCore 中，你既可以直接执行原始的 SQL 语句，也可以将其作为 SQLAlchemy Core 查询的一部分使用。它们返回的都是 ResultProxy，你可以继续与它交互，就像使用 SQLAlchemy Core 的 SQL 表达式语法构建的查询一样。建议你只在必要的时候使用原始查询和文本，因为使用它们有可能带来不可预见的结果和安全漏洞。首先，让我们尝试执行一个简单的 select 语句（见示例 2-30）。

示例 2-30　完全原始的查询

```
result = connection.execute("select * from orders").fetchall()
print(result)
```

结果如下：

```
[(1, 1, 0), (2, 2, 0)]
```

虽然我很少使用完全原始的 SQL 语句，但是经常使用小文本片段让查询变得更清晰。示例 2-31 是部分文本查询的例子，其中用到了 text() 函数。

示例 2-31　部分文本查询

```
from sqlalchemy import text
stmt = select([users]).where(text("username='cookiemon'"))
print(connection.execute(stmt).fetchall())
```

结果如下：

```
[(1, None, u'cookiemon', u'mon@cookie.com', u'111-111-1111', u'password ',
  datetime.datetime(2015, 3, 30, 13, 48, 25, 536450),
  datetime.datetime(2015, 3, 30, 13, 48, 25, 536457))
]
```

现在，你应该学会了如何使用 SQL 表达式语言处理 SQLAlchemy 中的数据。这一章，我们学习了创建、读取、更新和删除等操作。到这里，你可以停一停，多做一些尝试，进一步了解一下相关内容，然后再往下学。比如尝试创建更多 cookie、订单和行项目，使用查询链按订单和用户进行分组。当你对本章内容有了更深入的了解之后，就可以继续往下学习新内容了。接下来，让我们了解一下如何处理 SQLAlchemy 抛出的异常，以及如何使用事务对语句进行分组。

第 3 章

异常和事务

上一章中，做数据处理时，大都只用一个语句就完成了，并且回避了那些有可能导致错误的操作。在本章中，我们会故意执行一些不正确的操作，以便了解有哪些错误类型以及如何处理它们。本章最后，我们还会学习如何把需要成功执行的语句分组到事务中，以确保这些语句要么被正确执行，要么被正确清理掉。让我们先搞点破坏！

3.1 异常

SQLAlchemy 中可能发生的异常有很多，我们重点讲讲最常见的 `AttributeErrors` 和 `AttributeErrors`。掌握了这些常见异常的处理方法之后，相信对于那些不太常见的异常你也能轻松处理了。

为了学习本章，请启动一个新的 Python shell，并把在第 1 章创建的表加载进去。示例 3-1 再次把用到的表和连接列了出来，以供你参考。

示例 3-1　构建 shell 环境

```
from datetime import datetime

from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine,
                        CheckConstraint)
metadata = MetaData()

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
```

```
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2)),
    CheckConstraint('quantity > 0', name='quantity_positive')
)

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.now)
)

orders = Table('orders', metadata,
    Column('order_id', Integer()),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)

engine = create_engine('sqlite:///:memory:')
metadata.create_all(engine)
connection = engine.connect()
```

我们要学习的第一个错误是 AttributeError，它是我在调试代码的过程中最常遇见的错误。

3.1.1 AttributeError

我们从 AttributeError 学起。当试图访问一个不存在的属性时，就会出现 AttributeError 这个错误。如果你要访问的列在 ResultProxy 中不存在，经常会出现 AttributeError 错误。试图访问一个对象中不存在的属性时，就会引发 AttributeError 错误。在普通 Python 代码中，你可能遇到过这个错误。这里把它单独拿出来讲是因为它是 SQLAlchemy 中一个常见的错误，并且我们很容易忽略它发生的原因。为了引发这个错误，我们先向 users 表中插入一条记录并查询它。然后，尝试访问一个列，这个列在 users 表中存在，但在查询中不存在（见示例 3-2）。

示例 3-2　引发 AttributeError

```
from sqlalchemy import select, insert
ins = insert(users).values(
```

```
        username="cookiemon",
        email_address="mon@cookie.com",
        phone="111-111-1111",
        password="password"
    )
    result = connection.execute(ins) ❶

    s = select([users.c.username])
    results = connection.execute(s)
    for result in results:
        print(result.username)
        print(result.password) ❷
```

❶ 插入一条记录供测试。

❷ 查询结果中不存在 `password` 列，因为我们只查询了 `username` 列。

示例 3-2 中的代码导致 Python 抛出一个 `AttributeError` 并停止程序的执行。我们看看错误输出，了解一下发生了什么（见示例 3-3）。

示例 3-3　执行示例 3-2 产生的错误输出

```
cookiemon

AttributeError                          Traceback (most recent call last) ❶
<ipython-input-37-c4520631a10a> in <module>()
      3 for result in results:
      4     print(result.username)
----> 5     print(result.password) ❷

AttributeError: Could not locate column in row for column 'password' ❸
```

❶ 这一行指明了错误类型，并显示了错误跟踪。

❷ 这一行导致了错误的发生。

❸ 这是我们需要关注的部分。

示例 3-3 呈现的是 Python 中 `AttributeError` 的标准格式。第一行指明错误类型。接着是错误跟踪，指出错误发生的位置。由于我们的代码试图访问查询结果中一个不存在的列，所以它就把实际引发错误的代码行指了出来。最后一行提供了有关错误的重要细节。它再次指明错误类型以及发生错误的原因。本例中，引发 `AttributeError` 的原因是 `ResultProxy` 中的行不包含 `password` 列，因为我们只查询了用户名。`AttributeError` 是我们使用 SQLAlchemy 对象时经常会遇到的错误。此外，还有一些 SQLAlchemy 特有的错误，它们大都是由于错用 SQLAlchemy 语句引起的。下面来看一个例子：`IntegrityError`。

3.1.2　IntegrityError

另一个常见的 SQLAlchemy 错误是 `IntegrityError`，当试图做一些违反列约束或表约束的事情时，就会发生这种错误。这种类型的错误通常出现在你的操作破坏了唯一性约束的情

形下。比如，如果你试图创建的两个用户拥有相同的用户名，就会抛出 `IntegrityError`，因为 `users` 表中的用户名必须是唯一的。示例 3-4 中的代码会引发 `IntegrityError` 错误。

示例 3-4　引发 `IntegrityError` 错误

```
s = select([users.c.username])
connection.execute(s).fetchall() ❶

[(u'cookiemon',)]

ins = insert(users).values(
    username="cookiemon",
    email_address="damon@cookie.com",
    phone="111-111-1111",
    password="password"
)
result = connection.execute(ins) ❷
```

❶ 查看 `users` 表中的当前记录。

❷ 尝试插入第二条记录，这会导致发生错误。

示例 3-4 中的代码会让 SQLAlchemy 产生 `IntegrityError` 错误。接下来，让我们看看错误输出，并了解一下发生 `IntegrityError` 错误的原因是什么（见示例 3-5）。

示例 3-5　`IntegrityError` 错误输出

```
IntegrityError                            Traceback (most recent call last)
<ipython-input-7-6ecafb68a8ab> in <module>()
      5     password="password"
      6 )
----> 7 result = connection.execute(ins) ❶
... ❷

IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed:
users.username [SQL: u'INSERT INTO users (username, email_address, phone,
password, created_on, updated_on) VALUES (?, ?, ?, ?, ?, ?)'] [parameters:
('cookiemon', 'damon@cookie.com', '111-111-1111', 'password',
'2015-04-26 10:52:24.275082', '2015-04-26 10:52:24.275099')] ❸
```

❶ 这行触发了错误。

❷ 这里是长长的错误跟踪，我省略了。

❸ 这是需要我们关注的部分。

示例 3-5 展示了 SQLAlchemy 中 `IntegrityError` 错误的标准输出格式。第一行指明了错误类型。然后是错误跟踪详情。不过，这通常只是我们的执行语句和内部 SQLAlchemy 代码。对 `IntegrityError` 错误来说，我们通常完全可以忽略这些详情。最后一行包含了 `IntegrityError` 错误的重要细节，它再次指明了错误类型，然后指出是什么导致了这个错误。在本例中，导致错误的原因是：

```
UNIQUE constraint failed: users.username
```

这说明 users 表中的 username 列有唯一性约束，而我们的操作有悖于它。然后，它向我们提供了 SQL 语句以及编译参数的详细信息，类似于我们在第 2 章中看到的那样。由于这个错误，我们试图插入到 users 表中的新数据没有成功插入。这个错误还停止了程序的执行。

SQLAlchemy 中还有许多其他类型的错误，但上面这两种错误是最常见的。SQLAlchemy 中所有错误的输出格式都和我们上面看到的那两种格式一样。SQLAlchemy 文档中有关于其他类型错误的信息。

我们不希望程序在遇到错误时就崩溃，为此，我们还要学习如何正确地处理错误。

3.1.3 处理错误

为了防止错误导致程序崩溃或停止，需要正确地处理错误。我们可以像处理 Python 错误一样，使用 try/except 块。比如，可以使用 try/except 块来捕获错误并把错误消息打印出来，然后继续执行程序的其余部分。请看示例 3-6。

示例 3-6　捕获异常

```
from sqlalchemy.exc import IntegrityError ❶
ins = insert(users).values(
    username="cookiemon",
    email_address="damon@cookie.com",
    phone="111-111-1111",
    password="password"
)
try:
    result = connection.execute(ins)
except IntegrityError as error: ❷
    print(error.orig.message, error.params)
```

❶ sqlalchemy.exc 模块中所有的 SQLAlchemy 异常都可用。

❷ 捕获 IntegrityError 异常，这样才能访问这个异常的属性。

示例 3-6 中的语句与示例 3-4 一样，但是我们把 result = connection.execute(ins) 语句放在了 try/except 块中。当发生错误时，try/except 块会捕获 IntegrityError 并把错误信息和语句参数打印出来。在示例 3-6 中，我们在异常子句中打印错误消息，但其实我们可以在异常子句中编写任何我们喜欢的 Python 代码，比如在其中向用户返回一条错误消息，告知程序运行出现了故障。通过使用 try/except 块处理错误，可以让应用程序继续运行下去。

虽然示例 3-6 演示的是 IntegrityError 错误的处理方法，但是这种处理方法对 SQLAlchemy 产生的各种类型的错误都有效。有关其他 SQLAlchemy 异常的更多信息，请参阅 SQLAlchemy 文档。



请注意，`try/except` 块中的代码越少越好，并且只用它来捕获特定类型的错误。这样做可以防止捕获一些意料之外的错误，而这些错误和你要捕获的特定错误有不同的处理方式。

虽然可以使用传统的 Python 方法处理单个语句的异常，但是如果有多个数据库语句，并且它们之间相互依赖，那传统方法可能就不适用了。在这种情况下，需要把这些语句包装在一个数据库事务中。为此，SQLAlchemy 提供了一个简单易用的包装器——事务，它内置于连接对象中。

3.2 事务

我们不需要学习事务背后高深的数据库理论，只要把它看成一种打包方法就好。通过事务，我们可以把多个数据库语句打包成一组，作为一个整体，这组语句执行时要么成功要么失败。启动事务时，数据库系统会先记录数据库的当前状态，然后再执行多个 SQL 语句。如果事务中的所有 SQL 语句都成功执行，那么数据库将继续正常运行，并丢弃之前的数据库状态。图 3-1 描述了正常的事务流程。

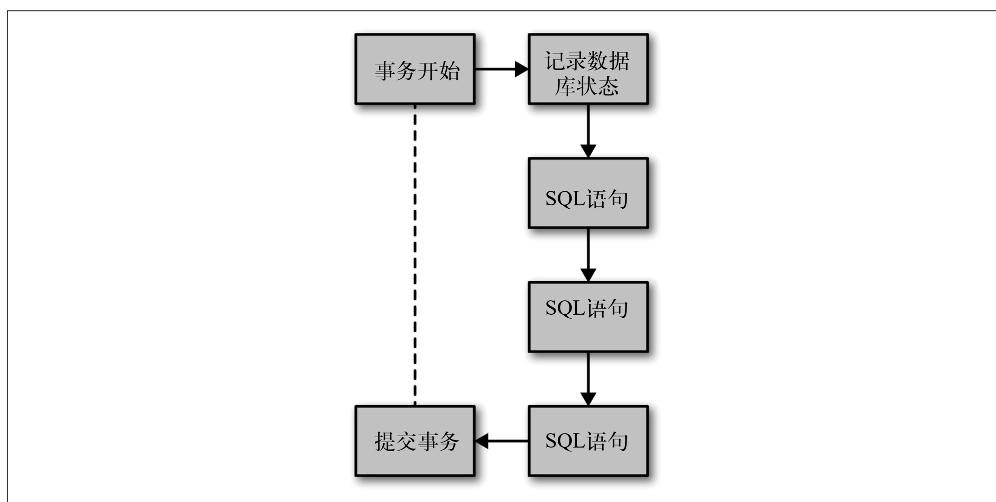


图 3-1：正常的事务流程

然而，只要事务中有一个或多个语句执行失败，我们就会捕获错误，并回滚到先前的状态。图 3-2 描述了事务失败时的处理流程。

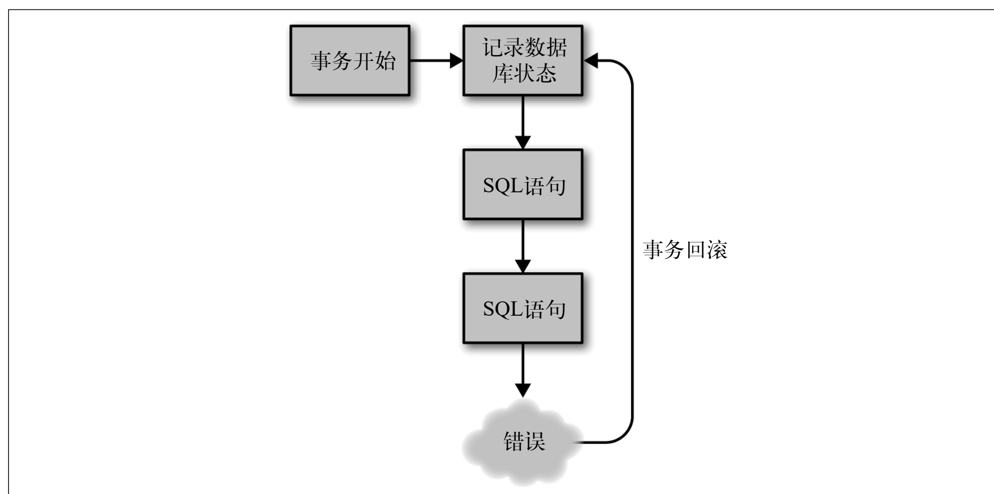


图 3-2：事务失败时的流程

下面举例说明一下什么时候需要在当前数据库中这样做。当一位客户跟我们订购了 cookie 之后，我们需要把指定的 cookie 发送给客户，然后从库存中删除。但是，如果我们没有足够的 cookie 来完成这笔订单呢？这时，我们需要先查看库存，而不是执行订单。可以使用事务来解决这个问题。

我们需要新开一个 Python shell，创建一些数据表。这里用到的表和第 2 章中差不多，但是，我们要向 `quantity` 列添加 `CheckConstraint`，这样可以确保数量不会小于 0，因为库存中的 cookie 不可能是负数。接下来，重新创建 cookiemon 用户以及 chocolate chip cookie 和 dark chocolate chip cookie 记录。把 chocolate chip cookie 的数量设置为 12 块，把 dark chocolate chip cookie 的数量设置为 1 块。示例 3-7 显示了创建带有 `CheckConstraint` 的数据表、添加 cookiemon 用户和添加 cookie 的完整代码。

示例 3-7　准备事务环境

```
from datetime import datetime

from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine,
                        CheckConstraint)
metadata = MetaData()

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2)),
    CheckConstraint('quantity >= 0', name='quantity_positive')
)
```

```
users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.now)
)

orders = Table('orders', metadata,
    Column('order_id', Integer()),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)

engine = create_engine('sqlite:///:memory:')
metadata.create_all(engine)
connection = engine.connect()
from sqlalchemy import select, insert, update
ins = insert(users).values(
    username="cookiemon",
    email_address="mon@cookie.com",
    phone="111-111-1111",
    password="password"
)
result = connection.execute(ins)
ins = cookies.insert()
inventory_list = [
    {
        'cookie_name': 'chocolate chip',
        'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe.html',
        'cookie_sku': 'CC01',
        'quantity': '12',
        'unit_cost': '0.50'
    },
    {
        'cookie_name': 'dark chocolate chip',
        'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe_dark.html',
        'cookie_sku': 'CC02',
        'quantity': '1',
        'unit_cost': '0.75'
    }
]
result = connection.execute(ins, inventory_list)
```

接下来，为 cookiemon 用户定义两个订单。第一个订单是 9 块 chocolate chip cookie，第二个订单是 4 块 chocolate chip cookie 和 1 块 dark chocolate chip cookie。我们使用上一章中学习的 `insert` 语句来完成这个任务。示例 3-8 是添加订单的代码。

示例 3-8　添加订单

```
ins = insert(orders).values(user_id=1, order_id='1')
result = connection.execute(ins)
ins = insert(line_items)
order_items = [
    {
        'order_id': 1,
        'cookie_id': 1,
        'quantity': 9,
        'extended_cost': 4.50
    }
]
result = connection.execute(ins, order_items) ❶

ins = insert(orders).values(user_id=1, order_id='2')
result = connection.execute(ins)
ins = insert(line_items)
order_items = [
    {
        'order_id': 2,
        'cookie_id': 1,
        'quantity': 4,
        'extended_cost': 1.50
    },
    {
        'order_id': 2,
        'cookie_id': 2,
        'quantity': 1,
        'extended_cost': 4.50
    }
]
result = connection.execute(ins, order_items) ❷
```

❶ 添加第一个订单。

❷ 添加第二个订单。

有了这些订单数据，我们就可以探索事务的工作原理了。我们需要定义一个名为 `ship_it` 的函数。这个函数接受一个 `order_id` 参数，它会从库存中删除 cookie，并把订单标记为“已发货”。示例 3-9 是 `ship_it` 函数的定义代码。

示例 3-9　定义 `ship_it()` 函数

```
def ship_it(order_id):

    s = select([line_items.c.cookie_id, line_items.c.quantity])
    s = s.where(line_items.c.order_id == order_id)
    cookies_to_ship = connection.execute(s)
```

```
    for cookie in cookies_to_ship: ❶
        u = update(cookies).where(cookies.c.cookie_id==cookie.cookie_id)
        u = u.values(quantity = cookies.c.quantity - cookie.quantity)
        result = connection.execute(u)
    u = update(orders).where(orders.c.order_id == order_id)
    u = u.values(shipped=True)
    result = connection.execute(u) ❷
    print("Shipped order ID: {}".format(order_id))
```

❶ 对于订单中的每种 cookie，我们都会从相应 cookies 表的 quantity 列中删除订单指定的数量，这样就可以知道还剩下多少。

❷ 更新订单，将其标记为“已发货”。

每当发出一个订单，ship_it 函数就会执行所有必需的操作。我们对第一个订单运行这个函数，然后查询 cookies 表，确保它正确地减少了 cookie 的数量。示例 3-10 展示了具体做法。

示例 3-10　对第一个订单运行 ship_it 函数

```
ship_it(1) ❶
s = select([cookies.c.cookie_name, cookies.c.quantity])
connection.execute(s).fetchall() ❷
```

❶ 对第一个 order_id 运行 ship_it 函数。

❷ 查看库存。

示例 3-10 的运行结果如下：

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

太好了！ ship_it 函数工作正常。从执行结果中可以看到，库存中没有足够的 cookie 来完成第二个订单。不过，在工作节奏很快的仓库中，这两个订单可能会被同时处理。现在尝试使用 ship_it 函数处理第二个订单，并观察会发生什么（如示例 3-11 所示）。

示例 3-11　对第二个订单运行 ship_it 函数

```
ship_it(2)
```

结果如下：

```
IntegrityError                            Traceback (most recent call last)
<ipython-input-9-47771be6653b> in <module>()
----> 1 ship_it(2)

<ipython-input-6-301c0ed7c4a1> in ship_it(order_id)
      7         u = update(cookies).where(cookies.c.cookie_id==cookie.cookie_id)
      8         u = u.values(quantity = cookies.c.quantity-cookie.quantity)
----> 9         result = connection.execute(u)
     10     u = update(orders).where(orders.c.order_id == order_id)
     11     u = u.values(shipped=True)
```

```
...

IntegrityError: (sqlite3.IntegrityError) CHECK constraint failed:
quantiuty_positive
[SQL: u'UPDATE cookies SET quantity=(cookies.quantity - ?) WHERE
cookies.cookie_id = ?'] [parameters: (4, 1)]
```

由于我们没有足够的 chocolate chip cookie 来完成订单，所以得到了一个 `IntegrityError` 错误。接下来使用示例 3-10 中的最后两行代码看看 `cookies` 表发生了什么变化：

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 0)]
```

由于 `IntegrityError` 错误，`ship_it` 函数没有去掉 chocolate chip cookie，而是去掉了 dark chocolate chip cookie。这是不对的！我们想向客户交付完整的订单，而不是订单的一部分。运用在 3.1.3 节中学到的 `try/except` 知识，我们可以设计出一个复杂的 `except` 方法来解决这种分批发货问题。但是，事务为我们提供了一种更好的方式来处理这类问题。

事务是通过调用连接对象的 `begin()` 方法启动的。调用 `begin()` 方法会返回一个事务对象，我们可以用它来控制所有语句的结果。如果所有语句都成功执行，就调用事务对象的 `commit()` 方法来提交事务。否则，就调用事务对象的 `rollback()` 方法进行回滚操作。下面重写 `ship_it` 函数，使用事务来保证语句安全地执行。示例 3-12 是重写后的 `ship_it` 函数。

示例 3-12　事务型 ship_it

```
from sqlalchemy.exc import IntegrityError ❶
def ship_it(order_id):
    s = select([line_items.c.cookie_id, line_items.c.quantity])
    s = s.where(line_items.c.order_id == order_id)
    transaction = connection.begin() ❷
    cookies_to_ship = connection.execute(s).fetchall() ❸

    try:
        for cookie in cookies_to_ship:
            u = update(cookies).where(cookies.c.cookie_id == cookie.cookie_id)
            u = u.values(quantity = cookies.c.quantity-cookie.quantity)
            result = connection.execute(u)
        u = update(orders).where(orders.c.order_id == order_id)
        u = u.values(shipped=True)
        result = connection.execute(u)
        print("Shipped order ID: {}".format(order_id))
        transaction.commit() ❹
    except IntegrityError as error:
        transaction.rollback() ❺
        print(error)
```

❶ 导入 `IntegrityError`，以便处理它。

❷ 启动事务。

❸ 获取所有结果，只是为了更容易地跟踪所发生的事。

❹ 若无错误发生，提交事务。

❺ 若有错误发生，回滚事务。

现在让我们把 chocolate chipcookie 的数量改回 1：

```
u = update(cookies).where(cookies.c.cookie_name == "dark chocolate chip")
u = u.values(quantity = 1)
result = connection.execute(u)
```

我们需要对第二个订单重新运行应用基于事务的 `ship_it` 函数。程序不会因为错误而停止，它会打印错误信息，并且不带错误跟踪详情。让我们使用示例 3-10 中的代码检查一下库存，确保它不会因发送一部分 cookie 而搞乱库存：

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

太棒了！我们的事务型函数没有搞乱库存或者导致应用程序崩溃。我们也不需要编写大量代码以便手动回滚到之前的状态。如你所见，事务在这种情况下非常有用，并且可以大大减少手动编码的工作量。

本章我们学习了单行语句和多行语句中的异常处理方法。通过在单行语句上添加普通的 `try/except` 块，可以防止应用程序在碰到数据库语句执行错误时发生崩溃。我们还学习了事务如何有助于避免数据库不一致问题，以及在一组语句组执行失败时如何防止程序崩溃。下一章中，我们将学习测试代码的方法，以便确保代码的行为符合我们的预期。

第 4 章

测试

应用程序测试大都包括单元测试和功能测试两部分。但是，在使用 SQLAlchemy 做单元测试时，要正确地模拟一条查询语句或者一个模型可能需要做很多工作。在对数据库做功能测试时，这些工作通常不会带来多大好处。所以，一般人们会为能在单元测试期间轻松模拟的查询创建包装器，或者在单元测试和功能测试中均只针对数据库做测试。我个人比较喜欢使用小巧的包装器函数，但是如果出于某种原因使得这样做没有意义，或者我处理的是遗留代码，那我会选择使用模拟方法。

本章讲解如何对数据库做功能测试，以及如何模拟 SQLAlchemy 查询和连接。

4.1 使用测试数据库做测试

在我们的示例应用程序中，有一个 app.py 文件，它包含应用程序逻辑；还有一个 db.py 文件，它包含数据库表和连接。你可以在 CH04/ 文件夹中找到这些文件。

应用程序的结构会对测试方式产生很大影响。在 db.py 文件中，你可以看到我们的数据库是通过 `DataAccessLayer` 类建立的。我们使用这个数据访问类来初始化数据库模式，并随时将其连接到引擎。在与 SQLAlchemy 结合使用的 Web 框架中，你会经常看到这种模式。`DataAccessLayer` 类初始化时，`dal` 变量中并不存在引擎和连接。示例 4-1 是 db.py 文件的一个片段。

示例 4-1 `DataAccessLayer` 类

```
from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine)
```

```
class DataAccessLayer:
    connection = None
    engine = None
    conn_string = None
    metadata = MetaData()
    cookies = Table('cookies',
                    metadata,
                    Column('cookie_id', Integer(), primary_key=True),
                    Column('cookie_name', String(50), index=True),
                    Column('cookie_recipe_url', String(255)),
                    Column('cookie_sku', String(55)),
                    Column('quantity', Integer()),
                    Column('unit_cost', Numeric(12, 2))
              ) ❶

    def db_init(self, conn_string): ❷
        self.engine = create_engine(conn_string or self.conn_string)
        self.metadata.create_all(self.engine)
        self.connection = self.engine.connect()

dal = DataAccessLayer() ❸
```

❶ 在完整的文件中，我们创建了所有表，而不只是 `cookies` 表。从第 1 章开始我们就一直在使用这些表。

❷ 这提供了一种方法，可以像工厂一样使用特定连接字符串初始化连接。

❸ 这创建了 `DataAccessLayer` 类的一个实例，你可以在应用程序中导入它。

接下来，为第 2 章中创建的 `get_orders_by_customer` 函数编写测试。你可以在 app.py 文件中找到这个函数。代码如示例 4-2 所示。

示例 4-2　待测试的 app.py

```
from db import dal ❶
from sqlalchemy.sql import select

def get_orders_by_customer(cust_name, shipped=None, details=False):
    columns = [dal.orders.c.order_id, dal.users.c.username, dal.users.c.phone]
    joins = dal.users.join(dal.orders) ❷

    if details:
        columns.extend([dal.cookies.c.cookie_name,
                        dal.line_items.c.quantity,
                        dal.line_items.c.extended_cost])
        joins = joins.join(dal.line_items).join(dal.cookies)

    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins).where(
        dal.users.c.username == cust_name)

    if shipped is not None:
        cust_orders = cust_orders.where(dal.orders.c.shipped == shipped)
```

```
    result = dal.connection.execute(cust_orders).fetchall()
```

❶ 这是 DataAccessLayer 类的实例，来自 db.py 文件。

❷ 因为数据表在 dal 对象中，所以我们从这里访问它们。

下面看看 get_orders_by_customer 函数的各种使用方法。在这个练习中，假设我们已经对函数的输入进行了验证，它们的类型都是正确的。不过，在你的测试中，明智的做法是确保你使用的数据有能正常工作的，也有可能会引起错误的。下面是函数的参数及其可能的取值。

- cust_name 可以是空的、包含有效客户名的字符串，或者是不包含有效客户名的字符串。
- shipped 可以是 None、True 或 False。
- details 可以是 Ture 或 False。

如果想测试所有可能的组合，那么需要 18（即 3 * 3 * 2）个测试来充分测试这个函数。



请注意，不要测试那些属于 SQLAlchemy 基本功能的代码，因为 SQLAlchemy 本身已经做了大量良好的测试。比如，我们不会测试简单的 insert、select、delete 或 update 语句，因为这些语句已经在 SQLAlchemy 项目中测试过了。我们应该测试代码操作的内容，这些内容可能会影响 SQLAlchemy 语句的运行方式或它返回的结果。

对于这个测试示例，我们会使用内置的 unittest 模块。如果你不熟悉这个模块，也不必担心，我会给大家讲讲重点内容。首先，我们需要编写测试类，并对 dal 的连接进行初始化。为此，新建一个名为 test_app.py 的文件，代码如例 4-3 所示。

示例 4-3　编写测试

```
import unittest

class TestApp(unittest.TestCase): ❶

    @classmethod
    def setUpClass(cls): ❷
        dal.db_init('sqlite:///:memory:') ❸
```

❶ unittest 需要的测试类继承自 unittest.TestCase。

❷ 在整个测试类中，setUpClass 方法只执行一次。

❸ 这行会初始化一个到内存数据库的连接，用来做测试。

接下来需要加载一些数据，以便在测试期间使用。这里不再给出完整的代码，因为其中用到的插入语句和第 2 章中使用的差不多，但为了方便使用 DataAccessLayer，我做了一些修改。你可以在 db.py 文件中找到完整的代码。当数据加载好之后，就可以动手编写一些测

试了。我们会把这些测试以函数的形式添加到 TestApp 类中，如示例 4-4 所示。

示例 4-4　针对空用户名的前 6 个测试

```
def test_orders_by_customer_blank(self): ❶
        results = get_orders_by_customer('')
        self.assertEqual(results, []) ❷

def test_orders_by_customer_blank_shipped(self):
    results = get_orders_by_customer('', True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped(self):
    results = get_orders_by_customer('', False)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_details(self):
    results = get_orders_by_customer('', details=True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_shipped_details(self):
    results = get_orders_by_customer('', True, True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped_details(self):
    results = get_orders_by_customer('', False, True)
    self.assertEqual(results, [])
```

❶ unittest 要求每个测试都以 test 打头。

❷ unittest 使用 assertEqual 来验证结果是否符合你的预期。由于没有找到用户，所以你会得到一个空列表。

现在，把测试文件保存为 test_app.py，然后使用如下命令运行单元测试：

```
# python -m unittest test_app
......

Ran 6 tests in 0.018s
```



你可能会得到一个有关 SQLite 和 decimal 类型的警告。忽略它就好，因为这在我们的例子中是正常的。之所以会发出警告，是因为 SQLite 中没有真正的 decimal 类型，而 SQLAlchemy 希望你知道，从 SQLite 的 float 类型转换而来可能有些奇怪。认真阅读这些消息是明智之举，因为在生产代码中，这些信息通常能为你指出正确的做事方法。这里我们特意触发了这个警告，以便让你看看它的样子。

接下来需要加载一些数据，并确保我们的测试仍然有效。同样，我们会重用第 2 章中的例子，插入相同的用户、订单和行项。不过，这次我们会把数据插入工作放入一个名为 prep_db 的函数中，这样就可以在测试之前通过调用 prep_db 这个函数来插入数据了。为了

简单起见，我把这个函数放入 db.py 文件中（参见示例 4-5）。但是，在真实情况下，通常会把它放到测试装置或实用程序文件中。

示例 4-5　插入一些测试数据

```
def prep_db():
    ins = dal.cookies.insert()
    dal.connection.execute(ins, cookie_name='dark chocolate chip',
            cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
            cookie_sku='CC02',
            quantity='1',
            unit_cost='0.75')
    inventory_list = [
        {
            'cookie_name': 'peanut butter',
            'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html',
            'cookie_sku': 'PB01',
            'quantity': '24',
            'unit_cost': '0.25'
        },
        {
            'cookie_name': 'oatmeal raisin',
            'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html',
            'cookie_sku': 'EWW01',
            'quantity': '100',
            'unit_cost': '1.00'
        }
    ]
    dal.connection.execute(ins, inventory_list)

    customer_list = [
        {
            'username': "cookiemon",
            'email_address': "mon@cookie.com",
            'phone': "111-111-1111",
            'password': "password"
        },
        {
            'username': "cakeeater",
            'email_address': "cakeeater@cake.com",
            'phone': "222-222-2222",
            'password': "password"
        },
        {
            'username': "pieguy",
            'email_address': "guy@pie.com",
            'phone': "333-333-3333",
            'password': "password"
        }
    ]
    ins = dal.users.insert()
    dal.connection.execute(ins, customer_list)
    ins = insert(dal.orders).values(user_id=1, order_id='wlk001')
    dal.connection.execute(ins)
```

```
        ins = insert(dal.line_items)
        order_items = [
            {
                'order_id': 'wlk001',
                'cookie_id': 1,
                'quantity': 2,
                'extended_cost': 1.00
            },
            {
                'order_id': 'wlk001',
                'cookie_id': 3,
                'quantity': 12,
                'extended_cost': 3.00
            }
        ]
        dal.connection.execute(ins, order_items)
        ins = insert(dal.orders).values(user_id=2, order_id='ol001')
        dal.connection.execute(ins)
        ins = insert(dal.line_items)
        order_items = [
            {
                'order_id': 'ol001',
                'cookie_id': 1,
                'quantity': 24,
                'extended_cost': 12.00
            },
            {
                'order_id': 'ol001',
                'cookie_id': 4,
                'quantity': 6,
                'extended_cost': 6.00
            }
        ]
        dal.connection.execute(ins, order_items)
```

到这里，`prep_db` 函数就创建好了。现在可以在 test_app.py 文件的 `setUpClass` 方法中调用它，把数据加载到数据库，再运行测试。`setUpClass` 方法的代码如下：

```
    @classmethod
    def setUpClass(cls):
        dal.db_init('sqlite:///:memory:')
        prep_db()
```

接下来，就可以使用这些测试数据来确保我们的函数在给定有效用户名时执行正确的操作。把测试以函数的形式添加到 `TestApp` 类中，如示例 4-6 所示。

示例 4-6　测试合法用户

```
    def test_orders_by_customer(self):
        expected_results = [(u'wlk001', u'cookiemon', u'111-111-1111')]
        results = get_orders_by_customer('cookiemon')
        self.assertEqual(results, expected_results)
```

```
    def test_orders_by_customer_shipped_only(self):
        results = get_orders_by_customer('cookiemon', True)
        self.assertEqual(results, [])

    def test_orders_by_customer_unshipped_only(self):
        expected_results = [(u'wlk001', u'cookiemon', u'111-111-1111')]
        results = get_orders_by_customer('cookiemon', False)
        self.assertEqual(results, expected_results)

    def test_orders_by_customer_with_details(self):
        expected_results = [
            (u'wlk001', u'cookiemon', u'111-111-1111', u'dark chocolate chip',
             2, Decimal('1.00')),
            (u'wlk001', u'cookiemon', u'111-111-1111', u'oatmeal raisin',
             12, Decimal('3.00'))
        ]
        results = get_orders_by_customer('cookiemon', details=True)
        self.assertEqual(results, expected_results)

    def test_orders_by_customer_shipped_only_with_details(self):
        results = get_orders_by_customer('cookiemon', True, True)
        self.assertEqual(results, [])

    def test_orders_by_customer_unshipped_only_details(self):
        expected_results = [
            (u'wlk001', u'cookiemon', u'111-111-1111', u'dark chocolate chip',
             2, Decimal('1.00')),
            (u'wlk001', u'cookiemon', u'111-111-1111', u'oatmeal raisin',
             12, Decimal('3.00'))
        ]
        results = get_orders_by_customer('cookiemon', False, True)
        self.assertEqual(results, expected_results)
```

使用示例 4-6 中的测试作为指导，你能完成针对不同用户（比如 `cakeeater`）的测试吗？能对系统中不存在的用户名进行测试吗？如果用户名是整数而不是字符串，结果会怎样？请将你的测试和示例代码中的测试进行比较，看看你的是否与本书中使用的测试相似。

到这里，我们已经学习了如何在功能测试中使用 SQLAlchemy 来判断一个函数在给定数据集上的行为是否与预期一致。我们还了解了如何设置 `unittest` 文件，以及如何准备要在测试中使用的数据库。接下来，我们可以在不访问数据库的情况下进行测试了。

4.2　使用mock

当在测试环境中创建测试数据库没有意义或者根本不可行时，mock 技术会派上大用场。如果你有大量逻辑需要对查询结果进行操作，那么模拟 SQLAlchemy 代码返回你想要的值会很有用，这样你就可以只测试周边逻辑。通常，当要模拟查询的某个部分时，我仍然会创建内存数据库，但是我不会向其中加载任何数据，而是会模拟数据库连接本身。这样我可以控制执行和获取方法返回的内容。我们将在本节中学习具体做法。

为了学习如何在测试中使用 mock，我们会为一个有效用户做一次测试。这一次我们将使用强大的 mock 库来控制连接返回的内容。在 Python 3 中，mock 是 `unittest` 模块的一部分。但是，如果你使用的是 Python 2，那么需要先使用 pip 来安装 mock 库，以便获得最新的功能。为此，可以使用以下命令：

```
pip install mock
```

安装好 mock 库后，就可以在测试中使用它了。mock 有一个 `patch()` 函数，它允许我们使用一个可以在测试中控制的 `MagicMock` 替换 Python 文件中给定的对象。`MagicMock` 是一个特殊的 Python 对象，它可以跟踪自身的使用方式，并允许我们根据其使用方式来定义它的行为。

首先，需要导入 mock 库。在 Python 2 中，需要使用如下代码：

```
import mock
```

在 Python 3 中，需要使用如下代码：

```
from unittest import mock
```

导入 mock 库之后，我们将把 `patch` 方法用作装饰器来替换 `dal` 对象的连接。装饰器函数用来包装另外一个函数，并可改变被包装函数的行为。因为 `dal` 对象是通过名称导入到 app.py 文件的，所以我们需要在 `app` 模块中包装它，然后将其作为参数传入测试函数。现在我们已经有了一个 mock 对象，接下来就可以为 `execute` 方法设置一个返回值了，本例中应该是什么也不返回，但要接上 `fetchall` 方法，它的返回值是我们想要测试的数据。示例 4-7 给出了使用 mock 代替 `dal` 对象所需的代码。

示例 4-7　模拟连接测试

```
import unittest
from decimal import Decimal

import mock

from db import dal, prep_db
from app import get_orders_by_customer

class TestApp(unittest.TestCase):
    cookie_orders = [(u'wlk001', u'cookiemon', u'111-111-1111')]
    cookie_details = [
        (u'wlk001', u'cookiemon', u'111-111-1111',
         u'dark chocolate chip', 2, Decimal('1.00')),
        (u'wlk001', u'cookiemon', u'111-111-1111',
         u'oatmeal raisin', 12, Decimal('3.00'))
    ]

    @mock.patch('app.dal.connection') ❶
    def test_orders_by_customer(self, mock_conn): ❷
```

```
        mock_conn.execute.return_value.fetchall.return_value = self.cookie_orders ❸
        results = get_orders_by_customer('cookiemon') ❹
        self.assertEqual(results, self.cookie_orders)
```

❶ 使用 mock 包装 app 模块中的 dal.connection。

❷ mock 以 mock_conn 的形式传入测试函数。

❸ 把 execute 方法的返回值设置成 fetchall 方法的返回值，并将 fetchall 方法的返回值设置为 self.cookie_order。

❹ 调用测试函数，模拟 dal.connection，并返回上一步设置的返回值。

你可以看到，使用复杂的查询或 ResultProxy（如示例 4-7 所示）尝试模拟完整的查询或连接时，可能很快就会变得非常乏味。但不要因此而不做，这对于发现未知 bug 非常有用。

如果你真的想模拟查询，也可以采用相同的方法，即使用 mock.patch 装饰器，并模拟 app 模块中的 select 对象。让我们使用一个空的查询尝试测试一下。我们必须模拟所有查询元素的返回值，在这个查询中是 select、select_from 和 where 子句。示例 4-8 演示了具体做法。

示例 4-8　模拟查询

```
    @mock.patch('app.select') ❶
    @mock.patch('app.dal.connection')
    def test_orders_by_customer_blank(self, mock_conn, mock_select): ❷
        mock_select.return_value.select_from.return_value.\
            where.return_value = '' ❸
        mock_conn.execute.return_value.fetchall.return_value = [] ❹
        results = get_orders_by_customer('')
        self.assertEqual(results, [])
```

❶ 查询链启动时，模拟 select 方法。

❷ 按顺序把装饰器传到函数中。从函数顺着装饰器栈向上，参数依次被添加到左边。

❸ 我们必须为链式查询的所有部分模拟返回值。

❹ 我们仍然需要模拟连接，否则 app 模块的 SQLAlchemy 代码将尝试进行查询。

希望大家把其余的测试作为练习，自己使用内存数据库和模拟测试类型来完成它们。此外，建议你把查询和连接都模拟一下，这样你会对整个模拟过程更加熟悉。

到这里，你应该已经熟悉如何测试包含 SQLAlchemy 函数的函数。你还学会了如何把数据预填充到测试数据库中，以便在测试中使用。最后，你还学习了如何模拟查询对象和连接对象。本章中举的例子比较简单，我们将在第 14 章中深入学习测试，届时会涉及 Flask、Pyramid 和 pytest。

接下来，我们将学习在 Python 中如何在不重建所有模式的情形下，通过反射使用 SQLAlchemy 处理现有数据库。

第 5 章
反射

使用反射技术可以利用现有数据库填充 SQLAlchemy 对象。你可以运用这种技术反射表、视图、索引和外键。本章将学习如何在示例数据库上使用反射。

为了测试，我建议你使用 Chinook 数据库。你可以在 http://chinookdatabase.codeplex.com/ 上了解到更多信息。这里，我们将使用 Chinook 数据库的 SQLite 版本，你可以在本书示例代码的 CH05/ 文件夹中找到它。这个文件夹中还包含一张数据库模式图像，通过它你可以清晰地了解本章中使用的模式。我们从反射单个表开始学起。

5.1　反射单个表

在第一个反射中，我们将生成 `Artist` 表。我们需要一个元数据对象来保存被反射的表模式信息，还需要一个引擎来连接到 Chinook 数据库。示例 5-1 是创建元数据和引擎的代码，这些代码你现在应该很熟悉了。

示例 5-1　创建初始对象

```
from sqlalchemy import MetaData, create_engine
metadata = MetaData()
engine = create_engine('sqlite:///Chinook_Sqlite.sqlite') ❶
```

❶ 这个连接字符串假设当前文件与示例数据库在同一个目录下。

当创建好元数据和引擎之后，我们就有了反射一个表所需要的一切。接下来，使用第 1 章中的表创建代码来创建一个表对象。但是，这一次我们不会手动定义列，而会使用 `autoload` 和 `autoload_with` 这两个关键字参数。这会把模式信息反射到 `metadata` 对象中，

并在 artist 变量中保存对表的引用。示例 5-2 演示了如何做反射。

示例 5-2　反射 Artist 表

```
from sqlalchemy import Table
artist = Table('Artist', metadata, autoload=True, autoload_with=engine)
```

最后一行就是反射 Artist 表所需的全部代码！现在我们就可以使用这个表了，就像在第 2 章中使用手动定义的表一样。示例 5-3 展示了如何使用简单的表查询来查看表中有哪些数据。

示例 5-3　使用 Artist 表

```
artist.columns.keys() ❶
from sqlalchemy import select
s = select([artist]).limit(10) ❷
engine.execute(s).fetchall()
```

❶ 显示列名。

❷ 查询前 10 条记录。

结果如下：

```
['ArtistId', 'Name']

[(1, u'AC/DC'),
 (2, u'Accept'),
 (3, u'Aerosmith'),
 (4, u'Alanis Morissette'),
 (5, u'Alice In Chains'),
 (6, u'Antônio Carlos Jobim'),
 (7, u'Apocalyptica'),
 (8, u'Audioslave'),
 (9, u'BackBeat'),
 (10, u'Billy Cobham')]
```


通过查看数据库模式，可以看到有一个与 Artist 表相关联的 Album 表。下面用相同的方法来反射它：

```
album = Table('Album', metadata, autoload=True, autoload_with=engine)
```

接下来查看一下 Album 表的元数据，看看都反射了些什么，代码如示例 5-4 所示。

示例 5-4　查看元数据

```
metadata.tables['album']

Table('album',
      MetaData(bind=None),
      Column('AlbumId', INTEGER(), table=<album>, primary_key=True, nullable=False),
      Column('Title', NVARCHAR(length=160), table=<album>, nullable=False),
```

```
        Column('ArtistId', INTEGER(), table=<album>, nullable=False),
              schema=None)
)
```

需要注意的是，指向 Artist 表的外键似乎并没有被反射。让我们查看一下 Album 表的 foreign_keys 属性，看看它是不是存在：

```
album.foreign_keys

set()
```

从这个结果看，外键的确没有被反射。之所以发生这种情况，是因为这两个表不是同时被反射的，而且在反射期间外键的目标不存在。为了不让你处于半断状态，SQLAlchemy 放弃了单向关系。可以使用第 1 章中学到的知识添加缺失的 ForeignKey，从而恢复它们之间的关系：

```
from sqlalchemy import ForeignKeyConstraint
album.append_constraint(
    ForeignKeyConstraint(['ArtistId'], ['artist.ArtistId'])
)
```

现在，重新运行示例 5-4，你会看到 ArtistId 列变成了一个 ForeignKey。

```
Table('album',
      MetaData(bind=None),
      Column('AlbumId', INTEGER(), table=<album>, primary_key=True,
             nullable=False),
      Column('Title', NVARCHAR(length=160), table=<album>, nullable=False),
      Column('ArtistId', INTEGER(), ForeignKey('artist.ArtistId'), table=<album>,
             nullable=False),
      schema=None)
```

接下来看看是否可以使用这个关系来正确地连接两个表。可以使用如下代码测试一下：

```
str(artist.join(album))

'artist JOIN album ON artist."ArtistId" = album."ArtistId"'
```

太好了！现在我们可以利用这种关系做查询了，其工作方式与 2.5 节中讨论的查询一样。

为数据库中的每个表重复这样的反射过程需要花费大量时间和精力。不过，幸运的是，SQLAlchemy 允许我们一次反射整个数据库。

5.2　反射整个数据库

为了反射整个数据库，可以使用 metadata 对象的 reflect 方法。reflect 方法会扫描引擎上的所有可用内容，并反射它所能反射的所有内容。下面，使用现有的 metadata 和 engine

对象来反射整个 Chinook 数据库：

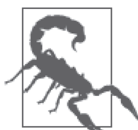
```
metadata.reflect(bind=engine)
```

如果这条语句执行成功，则不会返回任何内容。不过，可以运行如下代码来获取表名列表，以便查看 metadata 都反射了什么：

```
metadata.tables.keys()

dict_keys(['InvoiceLine', 'Employee', 'Invoice', 'album', 'Genre',
           'PlaylistTrack', 'Album', 'Customer', 'MediaType', 'Artist',
           'Track', 'artist', 'Playlist'])
```

从上面的结果可以看到，我们手动反射的表出现了两次，但它们的大小写不同。这是因为 SQLAlchemy 根据表名来反射表，而这些表名在 Chinook 数据库中是大写的。由于 SQLite 区分大小写，所以小写和大写名称指向了数据库中相同的表。



请小心：区分大小写有可能会在其他数据库（比如 Oracle）上造成错误。

至此，我们已经反射了整个数据库。接下来讲讲如何在查询中使用那些被反射的表。

5.3 使用反射对象构建查询

就像你在示例 5-3 中看到的那样，对于我们反射并存储在变量中的数据表，其查询方法与第 2 章中一样。但是，对于反射整个数据库时所反射的其他表，我们需要一种在查询中引用它们的方法。这可以通过把它们赋给 metadata.tables 属性的一个变量来实现，如示例 5-5 所示。

示例 5-5　在查询中使用反射表

```
playlist = metadata.tables['Playlist'] ❶

from sqlalchemy import select
s = select([playlist]).limit(10) ❷
engine.execute(s).fetchall()
```

❶ 创建一个引用 Playlist 表的变量 playlist。

❷ 在查询中使用 playlist 变量。

运行示例 5-5 中的代码，将得到如下结果：

```
engine.execute(s).fetchall()
[(1, 'Music'),
```

```
(2, 'Movies'),
(3, 'TV Shows'),
(4, 'Audiobooks'),
(5, '90's Music'),
(6, 'Audiobooks'),
(7, 'Movies'),
(8, 'Music'),
(9, 'Music Videos'),
(10, 'TV Shows')]
```

我们把想要使用的反射表赋给一个变量，然后就可以使用前几章中学到的方法来使用它们了。

反射是一个非常有用的工具。然而，在 SQLAlchemy 1.0 版本中，我们不能反射 `CheckConstraint`、注释或触发器。你也不能反射客户端的默认值，以及序列与列之间的关联。但是，可以使用第 1 章介绍的方法手动添加它们。

到这里，相信你已经学会如何反射单个表，以及如何修复这些表中缺失的关系等问题。此外，你还学会了如何反射整个数据库，以及如何在查询时使用其中的单个表。

本章介绍了 SQLAlchemy Core 和 SQL 表达式语言的基本内容。希望你认识到了 SQLAlchemy 功能的强大。相比于 SQLAlchemy ORM，SQLAlchemy Core 和 SQL 表达式语言经常被忽略，但是借助它们，我们可以为应用程序添加更多功能。接下来继续学习有关 SQLAlchemy ORM 的内容。

第二部分

SQLAlchemy ORM

提到 SQLAlchemy 时，大多数人首先想到的是 SQLAlchemy ORM。它为我们提供了一种将数据库模式和操作绑定到应用程序中使用的相同的数据对象上的非常有效的方法，还提供了一种快速构建应用程序并交付给客户的方法。ORM 使用起来非常简单，以至于大多数人都不曾考虑其代码可能产生的副作用。思考你的代码如何使用数据库仍然很重要。但是，ORM 的奇妙之处在于，你可以只在需要时才考虑这一点，而不必对每个查询都这样做。让我们开始使用 ORM 吧。

第 6 章

使用SQLAlchemy ORM定义模式

使用 SQLAlchemy ORM 时，定义的模式会略有不同，因为它关注的是用户定义的数据对象，而非底层数据库的模式。在 SQLAlchemy Core 中，我们会先创建一个元数据容器，然后声明一个与该元数据相关联的 `Table` 对象。而在 SQLAlchemy ORM 中，我们会定义一个类，它继承自一个名为 `declarative_base` 的特殊基类。`declarative_base` 把元数据容器和映射器（用来把类映射到数据表）结合在一起。如果类的实例已经保存，`declarative_base` 还会把类的实例映射到表中的记录。接下来，让我们深入学习这种定义表的方法。

6.1　使用ORM类定义表

ORM 使用的类应该满足如下四个要求。

- 继承自 `declarative_base` 对象。
- 包含 `__tablename__`，这是数据库中使用的表名。
- 包含一个或多个属性，它们都是 `Column` 对象。
- 确保一个或多个属性组成主键。

我们需要仔细检查最后两个与属性相关的需求。首先，在 ORM 类中定义列与在 `Table` 对象中定义列非常相似，相关内容已经在第 2 章中讲过。但是它们之间有一个非常重要的区别。在 ORM 类中定义列时，我们不必为 `Column` 构造函数提供列名作为第一个参数。相反，我们会把类的属性名赋给列名。除此之外，1.1 节和 1.3.1 节中讲解的所有其他内容同样适用于这里，并且能够按照预期工作。其次，对主键的要求乍看有点怪，但是，ORM 需要使用它来唯一地标识类的实例，并将其与底层数据库表中的特定记录关联起来。下面

看看定义成 ORM 类的 cookies 表（见示例 6-1）。

示例 6-1　定义成 ORM 类的 Cookies 表

```
from sqlalchemy import Table, Column, Integer, Numeric, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base() ❶

class Cookie(Base): ❷
    __tablename__ = 'cookies' ❸

    cookie_id = Column(Integer(), primary_key=True) ❹
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))
```

❶ 创建 declarative_base 类的一个实例。

❷ 继承 Base。

❸ 定义表名。

❹ 定义一个属性，并将其设置为主键。

上面代码得到的表和示例 2-1 中的表一样，可以通过查看 Cookie._table_ 属性来验证这一点：

```
>>> Cookie.__table__
Table('cookies', MetaData(bind=None),
    Column('cookie_id', Integer(), table=<cookies>, primary_key=True,
            nullable=False),
    Column('cookie_name', String(length=50), table=<cookies>),
    Column('cookie_recipe_url', String(length=255), table=<cookies>),
    Column('cookie_sku', String(length=55), table=<cookies>),
    Column('quantity', Integer(), table=<cookies>),
    Column('unit_cost', Numeric(precision=12, scale=2),
    table=<cookies>), schema=None)
```

接下来再看一个例子。这次我想根据示例 2-2 重新创建 users 表。示例 6-2 演示了附加关键字在 ORM 和 Core 模式中的使用方式是一样的。

示例 6-2　另一个拥有更多列的表

```
from datetime import datetime
from sqlalchemy import DateTime

class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True) ❶
```

```
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now) ❷
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now) ❸
```

❶ 这里，我们把该列设置为“必需”（`nullable=False`），并要求其值是唯一的。

❷ 如果没有指定日期，则默认将此列设置为当前时间。

❸ 通过使用 `onupdate`，使得每次更新记录时都将此列重置为当前时间。

键、约束、索引

1.3.2 节中讲到，除了在列本身中定义键和约束之外，还可以在表构造函数中定义键和约束。但是，使用 ORM 时，我们是在构建类，而不是使用表构造函数。在 ORM 中，可以通过类的 `__table_args__` 属性来添加它们。`__table_args__` 接收一个元组，里面包含多个表参数，如下所示：

```
class SomeDataClass(Base):
    __tablename__ = 'somedatatable'
    __table_args__ = (ForeignKeyConstraint(['id'], ['other_table.id']),
                      CheckConstraint(unit_cost >= 0.00',
                                      name='unit_cost_positive'))
```

从上面的代码看出，其语法与我们在 `Table()` 构造函数中使用的相同。这也适用于你在 1.3.3 节中学到的内容。

在本节中，我们学习了如何使用 ORM 定义两个表及其模式。定义数据模型的另一个重要部分是在多个表和对象之间建立关系。

6.2 关系

SQLAlchemy Core 和 SQLAlchemy ORM 的不同还体现在表和对象的关联上。ORM 使用类似的 `ForeignKey` 列来约束和链接对象。但是，它还使用 `relationship` 指令来提供一个属性，该属性可用于访问相关对象。这在使用 ORM 时的确会增加对数据库的使用和开销。但是，这种功能带来的好处远远多于缺点。我很想为你提供一个额外开销的粗略估计值，但这个值会因你的数据模型和模型的使用方式而不同。在大多数情况下，这个值甚至不值得考虑。示例 6-3 演示了如何使用 `relationship` 和 `backref` 方法定义一个关系。

示例 6-3　带关系的表

```
from sqlalchemy import ForeignKey
from sqlalchemy.orm import relationship, backref ❶
```

```
class Order(Base):
    __tablename__ = 'orders'

    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id')) ❷
    shipped = Column(Boolean(), default=False)

    user =  relationship("User", backref=backref('orders', order_by=id)) ❸
```

❶ 请注意，我们是如何从 `sqlalchem.orm` 中导入 `relationship` 和 `backref` 方法的。

❷ 定义一个 `ForeignKey`，就像我们在 SQLAlchemy Core 中所做的那样。

❸ 这建立了一对多的关系。

从 `Order` 类中定义的用户关系可以看出，它与 `User` 类建立了一对多的关系。可以通过访问 `user` 属性来获得与此 `Order` 相关的 `User`。这种关系还通过 `backref` 关键字参数在 `User` 类上建立了 `orders` 属性，`backref` 参数按照 `order_id` 排序。我们需要为 `relationship` 提供一个目标类，用来建立关系。你也可以向目标类添加一个反向引用。SQLAlchemy 知道如何使用我们定义的 `ForeignKey` 来匹配我们在关系中定义的类。在前面的示例中，`ForeignKey(`*`users.user_id`*`)`（拥有 `users` 表的 `user_id` 列）通过 `users` 的 `__tablename__` 属性映射到 `User` 类，从而形成关系。

还可以建立一对一的关系。在示例 6-4 中，`LineItem` 类与 `Cookie` 类就是一对一的关系。定义一对一的关系时，我们用到了 `uselist=False` 这个关键字参数。我们还使用了一个更简单的反向引用，因为我们不想控制顺序。

示例 6-4　带有关系的更多表

```
class LineItems(Base):
    __tablename__ = 'line_items'

    line_items_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                              order_by=line_items_id))

    cookie = relationship("Cookie", uselist=False) ❶
```

❶ 这里形成了一对一的关系。

这两个例子是学习关系的一个良好的起点。不过，关系是 ORM 中非常强大的一部分。这里只是做了简单介绍，第 14 章会进一步学习相关内容。在定义完所有类并建立好关系之后，就可以在数据库中创建表了。

6.3 模式持久化

为了创建数据库表，我们会用到 `Base` 实例中 `metadata` 的 `create_all` 方法。这个方法需要一个引擎实例，与 SQLAlchemy Core 中一样：

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///:memory:')

Base.metadata.create_all(engine)
```

在学习如何通过 ORM 处理类和表之前，需要先了解在 ORM 中是如何使用会话的。下一章会讲解这方面的内容。

第7章

使用SQLAlchemy ORM处理数据

我们已经定义好了表示数据库表的类并进行了持久化，接下来开始使用这些类来处理数据。本章将讲解如何插入、检索、更新和删除数据，还会学习如何对数据进行排序和分组，并了解关系是如何工作的。我们先从 SQLAlchemy 会话开始学起，它是 SQLAlchemy ORM 最重要的部分之一。

7.1　会话

会话是 SQLAlchemy ORM 和数据库交互的方式。它通过引擎包装数据库连接，并为通过会话加载或与会话关联的对象提供标识映射（identity map）。标识映射是一种类似于缓存的数据结构，它包含由对象表和主键确定的一个唯一的对象列表。会话还包装了一个事务，这个事务将一直保持打开状态，直到会话提交或回滚，这与 3.2 节中描述的过程很相似。

为了创建新会话，SQLAlchemy 提供了 sessionmaker 类，这个类可以确保在整个应用程序中能够使用相同的参数创建会话。sessionmaker 类通过创建一个 Session 类来实现这一点，Session 类是根据传递给 sessionmaker 工厂的参数配置的。sessionmaker 工厂在应用程序全局作用域中应该只使用一次，并且被视为配置设置。下面创建一个与内存中的 SQLite 数据库相关联的新会话：

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker ❶

engine = create_engine('sqlite:///:memory:')
```

```
Session = sessionmaker(bind=engine) ❷

session = Session() ❸
```

❶ 导入 sessionmaker 类。

❷ 使用 sessionmaker 提供的 bind 配置定义 Session 类。

❸ 使用生成的 Session 类创建 session，供我们使用。

现在已经有了一个可用来与数据库交互的 session。虽然 session 拥有连接数据库所需的一切，但在要求它之前，它是不会自行连接到数据库的。本章会继续拿第 6 章中创建的类作为示例。这里，我们还会向类中添加一些 __repr__ 方法，以方便查看和重建对象实例。不过，这些方法并不是必需的：

```
from datetime import datetime

from sqlalchemy import (Table, Column, Integer, Numeric, String, DateTime,
                        ForeignKey)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, backref


Base = declarative_base()


class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer(), primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __repr__(self): ❶
        return "Cookie(cookie_name='{self.cookie_name}', " \
                       "cookie_recipe_url='{self.cookie_recipe_url}', " \
                       "cookie_sku='{self.cookie_sku}', " \
                       "quantity={self.quantity}, " \
                       "unit_cost={self.unit_cost})".format(self=self)


class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
```

```
    updated_on = Column(DateTime(), default=datetime.now,
                        onupdate=datetime.now)
    def __repr__(self):
        return "User(username='{self.username}', " \
                     "email_address='{self.email_address}', " \
                     "phone='{self.phone}', " \
                     "password='{self.password}')".format(self=self)

class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id'))

    user = relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
                      "shipped={self.shipped})".format(self=self)


class LineItems(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))
    order = relationship("Order", backref=backref('line_items',
                         order_by=line_item_id))
    cookie = relationship("Cookie", uselist=False, order_by=id)

    def __repr__(self):
        return "LineItems(order_id={self.order_id}, " \
                          "cookie_id={self.cookie_id}, " \
                          "quantity={self.quantity}, " \
                          "extended_cost={self.extended_cost})".format(
                    self=self)


Base.metadata.create_all(engine) ❷
```

❶ `__repr__` 方法定义了应该如何表示对象。当调用构造函数创建并打印实例时，这个方法会被调用。它返回的内容将在稍后的打印输出中显示。

❷ 创建引擎定义的数据库表。

重新创建好类之后，该学习如何处理数据库中的数据了。让我们从插入数据开始学起吧。

7.2 插入数据

为了在数据库中创建一条新的 cookie 记录，先初始化 Cookie 类的一个新实例，这个实例中包含所需的数据。然后，把 Cookie 对象的新实例添加到会话中并提交会话。这很容易，

因为可以使用从 declarative_base 继承来的默认构造函数（见示例 7-1）。

示例 7-1　插入单个对象

```
cc_cookie = Cookie(cookie_name='chocolate chip', ❶
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=12,
                   unit_cost=0.50)
session.add(cc_cookie) ❷
session.commit() ❸
```

❶ 创建 Cookie 类的一个实例。

❷ 将实例添加到会话。

❸ 提交会话。

当调用会话的 commit() 方法时，cookie 就会被插入到数据库中。它还会使用数据库中记录的主键更新 cc_cookie。可以使用如下代码进行查看：

```
print(cc_cookie.cookie_id)
```

```
1
```

让我们花点时间了解一下，当运行示例 7-1 中的代码时，数据库都发生了什么变化。当我们创建 Cookie 类的实例并将其添加到会话中时，其实不会向数据库发送任何内容。直到调用会话的 commit() 方法，才会把内容发送到数据库。当调用 commit() 时，会发生以下事情：

```
INFO:sqlalchemy.engine.base.Engine:BEGIN (implicit) ❶

INFO:sqlalchemy.engine.base.Engine:INSERT INTO cookies (cookie_name,
cookie_recipe_url, cookie_sku, quantity, unit_cost) VALUES (?, ?, ?, ?, ?) ❷

INFO:sqlalchemy.engine.base.Engine:('chocolate chip',
'http://some.aweso.me/cookie/recipe.html', 'CC01', 12, 0.5) ❸

INFO:sqlalchemy.engine.base.Engine:COMMIT ❹
```

❶ 启动事务。

❷ 把记录插入到数据库。

❸ 插入的值。

❹ 提交事务。



如果想了解具体细节，可以把 echo=True 作为关键字参数添加到 create_engine 语句中，但是注意要把它放到连接字符串的后面。这个参数只在测试中使用，在实际软件产品中，请一定不要添加这个参数。

首先，启动一个新事务，把记录插入数据库。然后，引擎发送 insert 语句的值。最后，提交事务到数据库，并关闭事务。这种处理方法通常被称为“工作单元”模式。

接下来看看几种插入多条记录的方法。如果打算插入对象之后再对它们做一些其他的处理，那么需要使用示例 7-2 中提到的方法。这里只是简单地创建两个实例，而后把它们添加到会话中，然后再提交会话。

示例 7-2　插入多条记录

```
dcc = Cookie(cookie_name='dark chocolate chip',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
             cookie_sku='CC02',
             quantity=1,
             unit_cost=0.75)
mol = Cookie(cookie_name='molasses',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
             cookie_sku='MOL01',
             quantity=1,
             unit_cost=0.80)
session.add(dcc) ❶
session.add(mol) ❷
session.flush() ❸
print(dcc.cookie_id)
print(mol.cookie_id)
```

❶ 添加 dark chocolate chip cookie。

❷ 添加 molasses cookie。

❸ 刷新会话。

请注意，在示例 7-2 中，我们使用了会话的 flush() 方法而不是 commit() 方法。flush() 方法和 commit() 方法相似，但是它不会执行数据库提交并结束事务。因此，dcc 和 mol 实例仍然和会话连接，你可以使用它们执行更多数据库任务，而无须另外发起数据库查询。虽然我们要向数据库中添加多个记录，但只调用 session.flush() 语句一次就够了。调用 flush() 方法会把两条插入语句放入一个事务中，并发送到数据库。执行示例 7-2 中的代码，会得到如下结果：

```
2
3
```

当你想往数据表中插入数据，并且不需要对数据做额外处理时，使用 flush() 方法会更好，它允许你一次性向数据库插入多条记录。和示例 7-2 中使用的方法不同，示例 7-3 中的方法不会把记录与会话关联起来。

示例 7-3　批量插入多条记录

```
c1 = Cookie(cookie_name='peanut butter',
            cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
            cookie_sku='PB01',
```

```
                quantity=24,
                unit_cost=0.25)
c2 = Cookie(cookie_name='oatmeal raisin',
                cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
                cookie_sku='EWW01',
                quantity=100,
                unit_cost=1.00)
session.bulk_save_objects([c1,c2]) ❶
session.commit()
Print(c1.cookie_id)
```

❶ 添加 cookie 至列表，并使用 `bulk_save_objects` 方法批量保存。

执行示例 7-3 不会将任何内容打印到屏幕上，因为 `c1` 对象与会话没有关联，并且无法刷新 `cookie_id` 进行打印。如果查看发送到数据库的内容，可以看到事务中只有一条插入语句：

```
INFO:sqlalchemy.engine.base.Engine:INSERT INTO cookies (cookie_name,
cookie_recipe_url, cookie_sku, quantity, unit_cost) VALUES (?, ?, ?, ?, ?) ❶

INFO:sqlalchemy.engine.base.Engine:(
('peanut butter', 'http://some.aweso.me/cookie/peanut.html', 'PB01', 24, 0.25),
('oatmeal raisin', 'http://some.okay.me/cookie/raisin.html', 'EWW01', 100, 1.0))
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

❶ 单个插入。

相比于示例 7-2 中使用多条 `add` 语句和插入语句的做法，示例 7-3 中的方法执行起来要快得多。但是，这种速度的提升是以牺牲在正常添加和提交中可使用的一些特性为代价的，例如：

- 关系设置和操作得不到遵守或触发；
- 对象没有连接到会话；
- 默认情况下，不获取主键；
- 不会触发任何事件。

除了 `bulk_save_objects` 方法之外，还有其他多种通过字典创建和更新对象的方法，你可以在 SQLAlchemy 文档中了解更多有关批量操作及其性能的内容。



如果你想插入多条记录，并且不需要访问关系或插入的主键，请使用 `bulk_save_objects` 或相关方法。尤其是从外部数据源（例如 CSV 或带有嵌套数组的大型 JSON 文档）获取数据时，强烈建议你这样做。

现在，`cookies` 表中已经有了一些数据。接下来学习如何查询表以及获取这些数据。

7.3 查询数据

构建查询时要用到会话实例的 `query()` 方法。首先，把 `Cookie` 类传递给 `query()` 方法，这样就能获取 `cookies` 表中的所有记录，如示例 7-4 所示。

示例 7-4　获取所有 cookie

```
cookies = session.query(Cookie).all() ❶
print(cookies)
```

❶ 返回一个 Cookie 实例列表，代表 cookies 表中的所有记录。

示例 7-4 的输出结果如下：

```
[
    Cookie(cookie_name='chocolate chip',
           cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
           cookie_sku='CC01', quantity=12, unit_cost=0.50),
    Cookie(cookie_name='dark chocolate chip',
           cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
           cookie_sku='CC02', quantity=1, unit_cost=0.75),
    Cookie(cookie_name='molasses',
           cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
           cookie_sku='MOL01', quantity=1, unit_cost=0.80),
    Cookie(cookie_name='peanut butter',
           cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
           cookie_sku='PB01', quantity=24, unit_cost=0.25),
    Cookie(cookie_name='oatmeal raisin',
           cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
           cookie_sku='EWW01', quantity=100, unit_cost=1.00)
]
```

因为返回值是一个对象列表，所以我们可以像往常一样使用这些对象。由于这些对象与会话相连，所以我们可以更改或删除它们，并将更改持久化到数据库中。相关内容后面会讲解。

除了一次性获取所有对象之外，还可以将查询用作可迭代对象来遍历查询结果，如示例 7-5 所示。

示例 7-5　将查询用作可迭代对象

```
for cookie in session.query(Cookie): ❶
    print(cookie)
```

❶ 将查询用作可迭代对象时不调用 all()。

使用迭代方法的过程中，可以分别和每个记录对象交互，释放某个对象，并获得下一个对象。

运行示例 7-5，会得到如下结果：

```
Cookie(cookie_name='chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantity=12, unit_cost=0.50),
Cookie(cookie_name='dark chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
       cookie_sku='CC02', quantity=1, unit_cost=0.75),
```

```
    Cookie(cookie_name='molasses',
           cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
           cookie_sku='MOL01', quantity=1, unit_cost=0.80),
    Cookie(cookie_name='peanut butter',
           cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
           cookie_sku='PB01', quantity=24, unit_cost=0.25),
    Cookie(cookie_name='oatmeal raisin',
           cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
           cookie_sku='EWW01', quantity=100, unit_cost=1.00)
```

除了将查询用于迭代和调用其 `all()` 方法外，还有许多其他访问数据的方法。可以使用以下方法获取结果。

`first()`

若有，则返回第一个记录对象。

`one()`

查询所有行，如果返回的不是单个结果，则抛出异常。

`scalar()`

返回第一个结果的第一个元素。若无结果，返回 `None`；若结果多于一个，则引发错误。

最佳实践

编写生产代码时，应该遵循如下指导方针。

- 使用迭代而非 `all()` 方法获取记录。它比处理完整的对象列表更节省内存，而且我们往往一次只处理一条记录。
- 要获取单条记录，请使用 `first()` 方法，而非 `one()` 或 `scalar()` 方法，因为其他程序员同事更容易理解它。但是，如果你确定查询中有且只有一个结果，那请使用 `one()` 方法。
- 请谨慎使用 `scalar()` 方法，因为如果查询返回的数据不只一行一列，它就会引发错误。当查询获取的是全部记录时，该方法会返回整个记录对象，这会造成混淆并导致错误。

每次执行前面的示例代码查询数据库时，返回的每条记录都包含所有列。我们在工作中通常只需要用到这些列的一部分。如果这些额外列中的数据很大，应用程序的速度就会被拖慢，而且消耗的内存远远超过预期。尽管 SQLAlchemy 不会为查询或对象添加大量开销，但是，如果查询占用了太多内存，首先要考虑查询返回的数据是否有问题。接下来看看如何对查询返回的列数进行限制。

7.3.1　控制查询中的列数

为了限制查询返回的列数，需要把要查询的列传递给 query() 方法，各个列之间用逗号分隔。例如，你想运行一个查询，这个查询只返回 cookie 的名称和数量，如示例 7-6 所示。

示例 7-6　只查询 cookie_name 和 quantity 两列

```
print(session.query(Cookie.cookie_name, Cookie.quantity).first()) ❶
```

❶ 从 cookies 表查询 cookie_name 和 quantity 两个列，并返回第一个结果。

运行示例 7-6，会得到如下结果：

```
(u'chocolate chip', 12)
```

从上面的结果可以看到，查询返回的是一个由指定列的列值组成的元组。

我们已经可以构建简单的查询语句了，接下来看看还可以做些什么来改变查询语句返回结果的方式。首先学习如何改变返回结果的顺序。

7.3.2　排序

查看示例 7-6 的所有结果，而不仅仅是第一条记录，你会发现数据并不是按照特定顺序排列的。如果希望返回的列表有特定的顺序，可以使用 order_by() 语句，如示例 7-7 所示。示例中，我们希望结果按照现有 cookie 的数量进行排序。

示例 7-7　按数量进行升序排列

```
for cookie in session.query(Cookie).order_by(Cookie.quantity):
    print('{:3} - {}'.format(cookie.quantity, cookie.cookie_name))
```

运行示例 7-7，会得到如下结果：

```
  1 - dark chocolate chip
  1 - molasses
 12 - chocolate chip
 24 - peanut butter
100 - oatmeal raisin
```

如果想按倒序或降序排列，请使用 desc() 语句。这个函数的作用是按降序方式包装要排序的特定列，如示例 7-8 所示。

示例 7-8　按照数量进行降序排列

```
from sqlalchemy import desc
for cookie in session.query(Cookie).order_by(desc(Cookie.quantity)): ❶
    print('{:3} - {}'.format(cookie.quantity, cookie.cookie_name))
```

❶ 我们把想要按照倒序排列的列放入 desc() 函数中。



desc() 函数还可以作为列对象（比如 cookie.c.quantity.desc()）的方法进行调用。但是，如果在长语句中这样使用，可能会造成阅读困难，所以我总是把 desc() 用作函数。

如果应用程序只需要返回结果的一部分，可以对返回的结果数量进行限制。下面来具体讲讲。

7.3.3 限制返回结果集的条数

在前面的示例中，我们使用 first() 方法仅获取一行记录。虽然 query() 提供了我们请求的那行，但查询实际运行时会访问所有结果，而不仅仅是单个记录。如果想对查询进行限制，可以使用切片符号（slice notation）让 limit 语句成为查询的一部分。比如，你的时间只够制作两批 cookie，你想知道应该制作哪两种 cookie，那么你可以使用前面带顺序的查询，并添加 limit 语句来返回最需要补充的那两种 cookie。示例 7-9 显示了具体做法。

示例 7-9　两种库存量最少的 cookie

```
query = session.query(Cookie).order_by(Cookie.quantity)[:2]
print([result.cookie_name for result in query]) ❶
```

❶ 运行查询并将返回的列表切片。对大型结果集来说，这种做法的效率可能会很低。

运行示例 7-9，会得到如下结果：

```
[u'dark chocolate chip', u'molasses']
```

除了使用切片符号外，还可以使用 limit() 语句。

示例 7-10　两种库存量最少的 cookie（使用 limit()）

```
query = session.query(Cookie).order_by(Cookie.quantity).limit(2)
print([result.cookie_name for result in query])
```

现在，你已经知道需要烤哪两种 cookie 了，你可能还想知道当前库存中还有多少 cookie。许多数据库都包含 SQL 函数，这些函数的设计目的是让某些操作可以直接在数据库服务器上使用，比如 SUM。接下来学习一下如何使用这些函数。

7.3.4 内置SQL函数和标签

SQLAlchemy 还可以使用后端数据库中的 SQL 函数。其中，两个很常用的数据库函数是 SUM() 和 COUNT()。要使用这两个函数，需要先导入 sqlalchemy.func 模块。这两个函数被包装在它们操作的列上。因此，要获得 cookie 的总数，可以使用示例 7-11 中的代码。

示例 7-11　统计 cookie 数量

```
from sqlalchemy import func
inv_count = session.query(func.sum(Cookie.quantity)).scalar() ❶
print(inv_count)
```

❶ 请注意，这里使用了 `scalar()`，它只返回第一个记录最左边的列。



我一般总会导入 `func` 整个模块，因为直接导入 `sum` 可能会引起问题，而且还容易和 Python 内置的 `sum` 函数混淆。

运行示例 7-11，会得到如下结果：

```
138
```

接下来，使用 `count` 函数查看一下 `cookies` 表中有几种 cookie（见示例 7-12）。

示例 7-12　统计库存中有几种 cookie

```
rec_count = session.query(func.count(Cookie.cookie_name)).first()
print(rec_count)
```

在示例 7-11 中，我们使用 `scalar()` 方法得到了单个值。与此不同，在示例 7-12 中，我们得到的是一个元组，因为我们使用 `first()` 方法换掉了 `scalar()` 方法。

```
(5,)
```

使用 `count()` 和 `sum()` 这类函数最终会返回元组或列名为 `count_1` 的结果。这些返回形式通常不是我们想要的。另外，如果查询中有多个统计，那我们必须知道它们在语句中的出现次数，并将其合并到列名，因此第四个 `count()` 函数将是 `count_4`。命名应该清晰、明确，特别是当周围有其他 Python 代码时。

不过，值得庆幸的是，SQLAlchemy 提供了 `label()` 这个函数来解决这个问题。示例 7-13 执行的查询与示例 7-12 一样，但是它通过调用 `label()` 函数为我们访问的列起了一个更有用的名字。

示例 7-13　重命名统计列

```
rec_count = session.query(func.count(Cookie.cookie_name) \
                          .label('inventory_count')).first() ❶
print(rec_count.keys())
print(rec_count.inventory_count)
```

❶ 在想更改的列对象上调用 `label()` 函数。

运行示例 7-13，会得到如下结果：

```
[u'inventory_count']
5
```

我们已经学习了如何限制从数据库返回的列或行的数量。接下来学习如何根据指定的条件对查询数据进行过滤。

7.3.5 过滤

过滤查询是通过在查询后面添加 `filter()` 语句完成的。典型的 `filter()` 子句包含一个列、一个运算符和一个值或列。可以把多个 `filter()` 子句接在一起使用，或者在单个 `filter()` 中添加多个采用逗号分隔的 `ClauseElement` 表达式，它们的作用就像传统 SQL 语句中的 AND 一样。在示例 7-14 中，我们将查找名为“chocolate chip”的 cookie。

示例 7-14 使用 `filter()` 过滤 cookie 名

```
record = session.query(Cookie).filter(Cookie.cookie_name == 'chocolate chip').first()
print(record)
```

运行示例 7-14，将打印出 chocolate chip cookie 的记录：

```
Cookie(cookie_name='chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantity=12, unit_cost=0.50)
```

除此之外，还有一个 `filter_by()` 方法，其工作方式和 `filter()` 方法类似，只是它没有明确要求把类作为过滤器表达式的一部分，而是使用属性关键字表达式，这些表达式来自于查询的主实体或最后一个连接到语句的实体。另外，`filter_by()` 使用的是关键字赋值而非布尔值。示例 7-15 执行的查询和示例 7-14 完全一样。

示例 7-15 使用 `filter_by()` 过滤 `cookie_name`

```
record = session.query(Cookie).filter_by(cookie_name='chocolate chip').first()
print(record)
```

还可以使用 `where` 语句来查找所有包含“chocolate”这个词的 cookie 名（见示例 7-16）。

示例 7-16 查找名字中包含“chocolate”的 cookie

```
query = session.query(Cookie).filter(Cookie.cookie_name.like('%chocolate%'))
for record in query:
    print(record.cookie_name)
```

运行示例 7-16，会得到如下结果：

```
chocolate chip
dark chocolate chip
```

在示例 7-16 中，我们在过滤器语句中把 `Cookie.cookie_name` 列作为一种 `ClauseElement` 来

过滤结果，并且调用了 `ClauseElement` 的 `like()` 方法做匹配。此外，还有很多其他方法可供选择，请参考表 2-1。

如果不使用任何一个 `ClauseElement` 方法，那么就要在过滤器子句中使用运算符。大多数运算符的工作方式和你预想的一样。接下来讲讲这些运算符之间存在的一些差异。

7.3.6　运算符

到目前为止，我们用来过滤数据的方法有两种：一是利用列是否等于某个值，二是使用 `ClauseElement` 的方法，比如 `like()`。此外，还可以使用许多常见的运算符来过滤数据。SQLAlchemy 对大多数标准 Python 操作符做了重载，包括所有标准的比较运算符 (`==`、`!=`、`<`、`>`、`<=`、`>=`)，它们的功能和在 Python 语句中完全一样。在与 `None` 比较时，`==` 运算符被重载为 `IS  NULL` 语句。算术运算符（`\+`、`-`、`*`、`/` 和 `%`）还可以用来对独立于数据库的字符串做连接处理，如示例 7-17 所示。

示例 7-17　使用 + 连接字符串

```
results = session.query(Cookie.cookie_name, 'SKU-' + Cookie.cookie_sku).all()
for row in results:
    print(row)
```

示例 7-17 的运行结果如下：

```
('chocolate chip', 'SKU-CC01')
('dark chocolate chip', 'SKU-CC02')
('molasses', 'SKU-MOL01')
('peanut butter', 'SKU-PB01')
('oatmeal raisin', 'SKU-EWW01')
```

运算符的另一个常见用法是根据多个列来计算值。在处理财务数据或统计数据的应用程序和报表中，你经常要这样做。示例 7-18 是计算库存价值的一个例子。

示例 7-18　计算各种 cookie 的库存价值

```
from sqlalchemy import cast ❶
query = session.query(Cookie.cookie_name,
                      cast((Cookie.quantity * Cookie.unit_cost),
                           Numeric(12,2)).label('inv_cost')) ❷
for result in query:
    print('{} - {}'.format(result.cookie_name, result.inv_cost))
```

❶ `cast()` 函数允许我们做类型转换。本例中，我们获取的结果类似于 6.0000000000 这个样子，通过强制类型转换，我们可以让它看起来像货币。在 Python 中，还可以使用 `print('{} - {:.2f}'.format(row.cookie_name, row.inv_cost))` 完成相同的任务。

❷ 这里再次使用 `label()` 函数对列进行重命名。如果不进行重命名，列就不会出现在 `result` 对象的 keys 中，因为没有名字可操作。

示例 7-18 生成的结果如下：

```
chocolate chip - 6.00
dark chocolate chip - 0.75
molasses - 0.80
peanut butter - 6.00
oatmeal raisin - 100.00
```

如果需要组合 `where` 语句，有两种方法可供我们选用，其中一种方法就是布尔运算符。

7.3.7　布尔运算符

SQLAlchemy 还支持 SQL 布尔运算符 AND、OR 和 NOT，它们用位运算符（`&`、`|` 和 `~`）来表示。受 Python 运算符优先级规则的影响，在使用 AND 或 OR 重载运算符时一定要特别小心。比如，`&` 比 `<` 优先级高，所以当你写 `A < B & C < D` 时，你想得到的是 `(A < B) &(C < D)`，而实际得到的却是 `A < (B&C) < D`。

我们经常希望用包含和排除的方式把多个 `where` 子句链接在一起，这可以通过使用连接词来完成。

7.3.8　连接词

为了获得所期望的效果，我们既可以把多个 `filter()` 子句链接在一起，也可以使用连接词来实现，而且使用连接词的可读性更好、功能性更强。我还喜欢使用连接词代替布尔运算符，因为连接词会让代码更具表现力。SQLAlchemy 的连接词有 `and_()`、`or_()` 和 `not_()`。这些连接词带有下划线，以便与内置的关键字区分开。如果想获取价格低于某个数且数量超过指定值的 cookie 列表，可以使用示例 7-19 中的代码。

示例 7-19　使用带有多个 `ClauseElement` 表达式的 `filter()` 做 AND 运算

```
query = session.query(Cookie).filter(
    Cookie.quantity > 23,
    Cookie.unit_cost < 0.40
)
for result in query:
    print(result.cookie_name)
```

`or_()` 函数的作用与 `and_()` 函数正好相反，只要记录满足其中一个条件，就会被选出来。如果想搜索库存中数量在 10 到 50 之间或名字中包含 chip 这个词的 cookie，可以使用示例 7-20 中的代码。

示例 7-20　使用 `or()` 连接词

```
from sqlalchemy import and_, or_, not_
query = session.query(Cookie).filter(
    or_(
        Cookie.quantity.between(10, 50),
```

```
            Cookie.cookie_name.contains('chip')
        )
    )
    for result in query:
        print(result.cookie_name)
```

运行示例 7-20，会得到如下结果：

```
chocolate chip
dark chocolate chip
peanut butter
```

`not_()` 函数的工作方式与其他连接词类似，用来选择那些与指定条件不匹配的记录。

现在我们已经可以轻松地查询数据了，接下来该学习如何更新现有数据了。

7.4 更新数据

`update()` 方法和前面用过的 `insert()` 方法很相似，它们的语法几乎完全一样，但是 `update()` 可以指定一个 `where` 子句，用来指出要更新哪些行。与插入语句一样，更新语句可以由 `update()` 函数或者表格的 `update()` 方法来创建。如果省略 `where` 子句，则表示要更新表中的所有行。

例如，假设你已经完成了库存所需巧克力饼干（chocolate chip cookies）的烘焙工作。在示例 7-21 中，我们先通过更新查询把烘焙好的巧克力饼干添加到当前库存中，再查看当前库存中巧克力饼干的数量。

示例 7-21　通过对象更新数据

```
query = session.query(Cookie) ❶
cc_cookie = query.filter(Cookie.cookie_name == "chocolate chip").first() ❷
cc_cookie.quantity = cc_cookie.quantity + 120
session.commit()
print(cc_cookie.quantity)
```

❶ 第 1 行和第 2 行使用生成方法构建语句。

❷ 这里，我们要获取对象。但是，如果你已经有对象了，可以直接编辑它，而不必再次查询了。

运行示例 7-21，会得到如下结果：

```
132
```

此外，也可以就地更新数据，而无须先获取对象（见示例 7-22）。

示例 7-22　就地更新数据

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "chocolate chip")
query.update({Cookie.quantity: Cookie.quantity - 20}) ❶

cc_cookie = query.first() ❷
print(cc_cookie.quantity)
```

❶ `update()` 方法在会话外部更新记录，并返回更新的行数。

❷ 这里重用 `query` 对象，因为它有我们需要的选择条件。

运行示例 7-22，会得到如下结果：

```
112
```

除了更新数据之外，某些时候，我们还想从表中删除某些数据。接下来学习如何删除数据。

7.5　删除数据

创建删除语句时，既可以使用 `delete()` 函数，也可以使用表（包含待删数据的表）的 `delete()` 方法。与 `insert()` 和 `update()` 不同，`delete()` 不接收值参数，只接收一个可选的 `where` 子句，用来指定删除范围（省略 `where` 子句表示删除表中的所有行）。请看示例 7-23。

示例 7-23　删除数据

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "dark chocolate chip")
dcc_cookie = query.one()
session.delete(dcc_cookie)
session.commit()
dcc_cookie = query.first()
print(dcc_cookie)
```

运行示例 7-23，会得到如下结果：

```
None
```

此外，也可以就地删除数据，而无须先获取待删对象（见示例 7-24）。

示例 7-24　删除数据

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "molasses")
query.delete()

mol_cookie = query.first()
print(mol_cookie)
```

运行示例 7-24，会得到如下结果：

```
None
```

现在，让我们综合运用前面学过的内容向 `users`、`orders` 和 `line_items` 表中加载一些数据。你可以直接复制下面这些代码进行添加，不过最好花点时间尝试使用不同的插入数据的方法。

```
cookiemon = User(username='cookiemon',
                 email_address='mon@cookie.com',
                 phone='111-111-1111',
                 password='password')
cakeeater = User(username='cakeeater',
                 email_address='cakeeater@cake.com',
                 phone='222-222-2222',
                 password='password')
pieperson = User(username='pieperson',
                 email_address='person@pie.com',
                 phone='333-333-3333',
                 password='password')
session.add(cookiemon)
session.add(cakeeater)
session.add(pieperson)
session.commit()
```

现在，我们有了客户，接下来开始把他们的订单和行项目输入到系统中。在把这两个对象插入到表中时，我们会利用它们之间的关系。如果想将一个对象关联到另一个对象，可以通过把它指派给关系属性来实现，就像我们对其他属性所做的那样。你会在示例 7-25 中多次看到这一点。

示例 7-25　添加相关对象

```
o1 = Order()
o1.user = cookiemon ❶
session.add(o1)

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "chocolate chip").one()
line1 = LineItem(cookie=cc, quantity=2, extended_cost=1.00) ❷

pb = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "peanut butter").one()
line2 = LineItem(quantity=12, extended_cost=3.00) ❸
line2.cookie = pb
line2.order = o1

o1.line_items.append(line1) ❹
o1.line_items.append(line2)

session.commit()
```

❶ 把 cookiemon 设置为下订单的用户。

❷ 为订单创建一个行项目，并与 cookie 关联。

❸ 一次构建一个行项目。

❹ 通过关系把行项目添加到订单中。

在示例 7-25 中，我们创建了一个空的 `Order` 实例，并将其 `user` 属性设置为 `cookiemon` 实例。接下来，将它添加到会话中。然后查询 chocolate chip cookie 并创建一个 `LineItem`，把 cookie 设置为刚才查询的 chocolate chip cookie。我们对订单的第二行项目重复这个过程，但要一步步地创建。最后，将行项目添加到订单中并提交订单。

下面为另一个用户再添加一个订单。

```
o2 = Order()
o2.user = cakeeater

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "chocolate chip").one()
line1 = LineItem(cookie=cc, quantity=24, extended_cost=12.00)

oat = session.query(Cookie).filter(Cookie.cookie_name ==
                                   "oatmeal raisin").one()
line2 = LineItem(cookie=oat, quantity=6, extended_cost=6.00)

o2.line_items.append(line1)
o2.line_items.append(line2)

session.add(o2) ❶
session.commit()
```

❶ 我将这一行与其他添加内容一起向下移动，SQLAlchemy 会确定创建它们的适当顺序，以确保成功。

第 6 章中，我们学习了如何定义 `ForeignKeys` 和关系，但是到目前为止，我们还没有用它们做过查询。下面就来看看这些关系。

7.6 连接

下面学习如何使用 `join()` 和 `outerjoin()` 方法来查询相关数据。例如，为了执行 cookiemon 客户所下的订单，我们需要确认每种 cookie 的订购量。总共需要使用 3 个连接才能获取 cookie 的名称。相比于原始的 SQL，使用 ORM 会让这样的查询变得更容易（见示例 7-26）。

示例 7-26　使用连接查询多个表

```
query = session.query(Order.order_id, User.username, User.phone,
                      Cookie.cookie_name, LineItem.quantity,
                      LineItem.extended_cost)
query = query.join(User).join(LineItem).join(Cookie) ❶
results = query.filter(User.username == 'cookiemon').all()
print(results)
```

❶ 告诉 SQLAlchemy 连接相关对象。

运行示例 7-26，会得到如下结果：

```
[
    (u'1', u'cookiemon', u'111-111-1111', u'peanut butter', 12, Decimal('3.00')),
    (u'1', u'cookiemon', u'111-111-1111', u'chocolate chip', 2, Decimal('1.00'))
]
```

此外，获取所有用户（包括那些当前没有订单的用户）的订单数量也很有用。为此，必须使用 `outerjoin()` 方法，并且要对连接顺序多加注意，因为应用 `outerjoin()` 方法的表将会返回所有结果（见示例 7-27）。

示例 7-27　使用外连接查询多个表

```
query = session.query(User.username, func.count(Order.order_id))
query = query.outerjoin(Order).group_by(User.username)
for row in query:
    print(row)
```

运行示例 7-27，会得到如下结果：

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieperson', 0)
```

到目前为止，我们一直在查询中使用和连接不同的表。但是，如果我们有一个“自引用”的表，比如员工和老板的关系表，该怎么办呢？ ORM 允许我们建立指向同一个表的关系。不过，我们需要指定一个名为 `remote_side` 的选项，来形成多对一的关系：

```
class Employee(Base):
    __tablename__ = 'employees'

    id = Column(Integer(), primary_key=True)
    manager_id = Column(Integer(), ForeignKey('employees.id'))
    name = Column(String(255), nullable=False)

    manager = relationship("Employee", backref=backref('reports'),
                           remote_side=[id]) ❶

Base.metadata.create_all(engine)
```

❶ 创建一个关系，反向引用同一个表，指定 `remote_side`，形成多对一的关系。

接下来，让我们添加两个雇员，其中一个是另一个的上司：

```
marsha = Employee(name='Marsha')
fred = Employee(name='Fred')

marsha.reports.append(fred)
```

```
session.add(marsha)
session.commit()
```

现在，如果想打印向 Marsha 报告的员工，可以通过访问 `reports` 属性来实现，如下所示：

```
for report in marsha.reports:
    print(report.name)
```

做数据报表时，数据分组很有用。接下来学习一下如何对数据进行分组。

7.7　分组

使用分组时，你需要一个或多个列来做分组，以及一个或多个列来做统计，比如计数、求和等，就像在普通 SQL 中所做的那样。下面，让我们按客户获得订单数量（见示例 7-28）。

示例 7-28　数据分组

```
query = session.query(User.username, func.count(Order.order_id)) ❶
query = query.outerjoin(Order).group_by(User.username) ❷
for row in query:
    print(row)
```

❶ 使用 `count()` 统计订单数。

❷ 根据用户名进行分组。

运行示例 7-28，会得到如下结果：

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieguy', 0)
```

在前面的例子中，构建语句时，我们已经用过生成方法。接下来详细讲讲。

7.8　链式调用

前面我们已经多次用过链式调用，只是没有明说。构建查询时，如果你想把逻辑清晰地表达出来，那么使用链式调用会特别有用。下面创建一个函数，让其为我们获取订单数据，代码如示例 7-29 所示。

示例 7-29　链式调用

```
def get_orders_by_customer(cust_name):
    query = session.query(Order.order_id, User.username, User.phone,
                          Cookie.cookie_name, LineItem.quantity,
                          LineItem.extended_cost)
    query = query.join(User).join(LineItem).join(Cookie)
```

```
    results = query.filter(User.username == cust_name).all()
    return results

get_orders_by_customer('cakeeater')
```

运行示例 7-29，会得到如下结果：

```
[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]
```

但是，如果我们只想获得那些已经发货或者还没有发货的订单，该怎么办呢？为此，必须编写其他函数来支持额外的过滤选项，或者使用条件来构建查询链。另外，我们可能还需要一个选项，用来指定是否包含细节。利用这种把查询和子句链在一起的方法，我们可以做出功能强大的报表，以及构建出复杂的查询（见示例 7-30）。

示例 7-30　带条件的链式调用

```
def get_orders_by_customer(cust_name, shipped=None, details=False):
    query = session.query(Order.order_id, User.username, User.phone)
    query = query.join(User)
    if details:
        query = query.add_columns(Cookie.cookie_name, LineItem.quantity,
                                  LineItem.extended_cost)
        query = query.join(LineItem).join(Cookie)
    if shipped is not None:
        query = query.filter(Order.shipped == shipped)
    results = query.filter(User.username == cust_name).all()
    return results

print(get_orders_by_customer('cakeeater')) ❶

print(get_orders_by_customer('cakeeater', details=True)) ❷

print(get_orders_by_customer('cakeeater', shipped=True)) ❸

print(get_orders_by_customer('cakeeater', shipped=False)) ❹

print(get_orders_by_customer('cakeeater', shipped=False, details=True))❺
```

❶ 获取所有订单。

❷ 获取所有订单细节。

❸ 只获取已经发货的订单。

❹ 获取还没发货的订单。

❺ 获取还没发货的订单细节。

运行示例 7-30，会得到如下结果：

```
[(u'2', u'cakeeater', u'222-222-2222')]

[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

[]

[(u'2', u'cakeeater', u'222-222-2222')]

[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]
```

到目前为止，本章示例中使用的都是 ORM。其实你也可以使用原始 SQL 语句，下面就来具体了解一下。

7.9 原始查询

虽然我很少使用完全原始的 SQL 语句，但是我经常使用小文本片段来让查询变得更清晰。示例 7-31 是使用原始 SQL `where` 子句的例子，其中用到了 `text()` 函数。

示例 7-31 使用 `text()` 函数

```
from sqlalchemy import text
query = session.query(User).filter(text("username='cookiemon'"))
print(query.all())
```

运行示例 7-31，会得到如下结果：

```
[User(username='cookiemon', email_address='mon@cookie.com',
      phone='111-111-1111', password='password')]
```

现在，你应该学会了如何使用 ORM 处理 SQLAlchemy 中的数据。我们学习了创建、读取、更新和删除等操作。到这里，你可以停一停，多做一些尝试，进一步了解一下相关内容，然后再往下学。比如尝试创建更多 cookie、订单和行项目，使用查询链按订单和用户进行分组。

当你对本章内容有了更为深入的了解之后，就可以继续往下学习新内容了。接下来，让我们了解一下如何处理 SQLAlchemy 抛出的异常，以及如何使用事务对语句进行分组。

第 8 章

理解会话和异常

上一章中我们学习了大量与会话相关的内容，并且回避了那些有可能引发异常的操作。本章，我们会进一步学习有关对象和 SQLAlchemy 会话交互的内容。本章最后，我们会故意执行一些错误操作，以便了解有哪些异常类型以及如何处理它们。我们先来学习更多与会话有关的内容。首先，使用第 6 章中的表创建 SQLite 内存数据库。

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Session = sessionmaker(bind=engine)

session = Session()

from datetime import datetime

from sqlalchemy import (Table, Column, Integer, Numeric, String, DateTime,
                        ForeignKey, Boolean)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, backref


Base = declarative_base()


class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
```

```
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __init__(self, name, recipe_url=None, sku=None, quantity=0,
                 unit_cost=0.00):
        self.cookie_name = name
        self.cookie_recipe_url = recipe_url
        self.cookie_sku = sku
        self.quantity = quantity
        self.unit_cost = unit_cost

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
                       "cookie_recipe_url='{self.cookie_recipe_url}', " \
                       "cookie_sku='{self.cookie_sku}', " \
                       "quantity={self.quantity}, " \
                       "unit_cost={self.unit_cost})".format(self=self)


class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now)

    def __init__(self, username, email_address, phone, password):
        self.username = username
        self.email_address = email_address
        self.phone = phone
        self.password = password

    def __repr__(self):
        return "User(username='{self.username}', " \
                     "email_address='{self.email_address}', " \
                     "phone='{self.phone}', " \
                     "password='{self.password}')".format(self=self)

class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id'))
    shipped = Column(Boolean(), default=False)

    user = relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
                      "shipped={self.shipped})".format(self=self)
```

```
class LineItem(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))
    cookie = relationship("Cookie", uselist=False)

    def __repr__(self):
        return "LineItems(order_id={self.order_id}, " \
                         "cookie_id={self.cookie_id}, " \
                         "quantity={self.quantity}, " \
                         "extended_cost={self.extended_cost})".format(
                    self=self)

Base.metadata.create_all(engine)
```

我们已经定义好了初始数据库和对象，接下来可以进一步学习会话如何与对象交互了。

8.1 SQLAlchemy会话

第 7 章中已经介绍过一些与 SQLAlchemy 会话相关的内容，这里重点讲讲数据对象是如何与会话交互的。

当使用查询获取对象时，会得到一个和会话相连的对象。这个对象会在会话的几个状态中切换。

会话状态

了解会话状态对于排除异常和处理异常行为很有用。数据对象实例有四种可能的状态。

transient（**瞬时状态**）

实例不在会话中，也不在数据库中。

pending（**挂起状态**）

实例由 `add()` 方法添加到会话中，但仍未刷新或提交。

persistent（**持久化状态**）

会话中的对象在数据库中有对应的记录。

detached（**脱管状态**）

实例不再与会话相连，但在数据库中仍然有一条记录。

当使用某个实例时，我们可以观察到它在这些状态之间转换。先来创建一个 Cookie 的实例：

```
cc_cookie = Cookie('chocolate chip',
                   'http://some.aweso.me/cookie/recipe.html',
                   'CC01', 12, 0.50)
```

为了查看实例的状态，可以使用 SQLAlchemy 提供的功能强大的 inspect() 方法。当在一个实例上应用 inspect() 方法时，可以访问一些有用的信息。下面检查一下 cc_cookie 实例：

```
from sqlalchemy import inspect
insp = inspect(cc_cookie)
```

本例中，我们关注的是表示当前状态的 transient、pending、persistent、detached 属性。让我们循环遍历这些属性，如示例 8-1 所示。

示例 8-1　获取实例的会话状态

```
for state in ['transient', 'pending', 'persistent', 'detached']:
    print('{:>10}: {}'.format(state, getattr(insp, state)))
```

运行示例 8-1 会得到一系列状态，其中布尔值表示哪个是当前状态。

```
 transient: True
   pending: False
persistent: False
  detached: False
```

从上面的输出可以看到，cookie 实例的当前状态是“瞬时状态”（transient），这是在刷新或提交到数据库之前新创建的对象所处的状态。如果在当前会话中添加 cc_cookie，然后重新运行示例 8-1，将得到以下输出：

```
 transient: False
   pending: True
persistent: False
  detached: False
```

现在，我们已经把 cookie 实例添加到当前会话中了，可以看到其当前状态由 transient 变成了 pending。如果提交会话，然后再次运行示例 8-1，就可以看到其状态又变成了 persistent：

```
 transient: False
   pending: False
persistent: True
  detached: False
```

最后，为了让 cc_cookie 进入脱管状态，需要调用会话的 expunge() 方法。当你想把数据从一个会话移到另一个会话时，可能需要这样做。而你希望把数据从一个会话移到另一个会话的一种情况是，你想把主要数据库中的数据归档或合并到数据仓库：

```
session.expunge(cc_cookie)
```

执行上面这行代码后，重新运行示例 8-1，你会看到 cc_cookie 进入了脱管状态。

```
 transient: False
   pending: False
persistent: False
  detached: True
```

在普通代码中使用 inspect() 方法时，你可能会使用 insp.transient、insp.pending、insp.persistent 和 insp.detached 获取各个状态的情况。而这里，我们使用 getattr() 方法来访问它们，这样就可以遍历这些状态，而不必分别对每个状态进行硬编码。

现在，我们已经知道一个对象如何在各种会话状态中切换。接下来看看如何使用 insp 在提交实例之前查看它的历史记录。首先，把对象添加回会话并更改 cookie_name 属性：

```
session.add(cc_cookie)
cc_cookie.cookie_name = 'Change chocolate chip'
```

接着，通过 insp 的 modifed 属性，查看实例对象是否发生了更改。

```
insp.modified
```

运行结果为 True。我们还可以使用 insp 的 attrs 集合来查看发生了什么变化，如示例 8-2 所示。

示例 8-2　打印属性更改历史

```
for attr, attr_state in insp.attrs.items():
    if attr_state.history.has_changes(): ❶
        print('{}: {}'.format(attr, attr_state.value))
        print('History: {}\n'.format(attr_state.history)) ❷
```

❶ 检查属性状态，看看会话是否能发现变化。

❷ 打印有更改的属性的 history 对象。

运行示例 8-2，会得到如下结果：

```
cookie_name: Change chocolate chip
History: History(added=['Change chocolate chip'], unchanged=(), deleted=())
```

从示例 8-2 的输出结果可以看到，cookie_name 发生了变化。当查看这个属性的历史记录时，可以看到该属性上添加或更改了什么。这帮助我们了解了对象是如何和会话交互的。接下来学习如何处理使用 SQLAlchemy ORM 的过程中出现的异常。

8.2　异常

SQLAlchemy 中可能发生的异常有很多，但这里重点讲讲 MultipleResultsFound 和

DetachedInstanceError 两种异常。这两个异常很常见，也是一组类似异常的一部分。学会了处理这些异常的方法，你就能轻松地处理遇到的其他异常了。

8.2.1 MultipleResultsFound异常

在使用 .one() 查询方法时，如果返回了多个结果，就会发生 MultipleResultsFound 异常。为了演示这个异常，我们需要先创建另外一个 cookie 并将其保存到数据库中：

```
dcc = Cookie('dark chocolate chip',
             'http://some.aweso.me/cookie/recipe_dark.html',
             'CC02', 1, 0.75)
session.add(dcc)
session.commit()
```

现在，数据库中已经有多个 cookie 了，接下来触发 MultipleResultsFound 异常（见示例 8-3）。

示例 8-3　触发 MultipleResultsFound 异常

```
results = session.query(Cookie).one()
```

运行示例 8-3 会发生异常，因为我们的数据源中有两个 cookie 与查询相匹配。异常会停止程序的执行。示例 8-4 显示了这个异常的信息。

示例 8-4　MultipleResultsFound 异常的信息

```
MultipleResultsFound                 Traceback (most recent call last) ❶
<ipython-input-20-d88068ecde4b> in <module>()
----> 1 results = session.query(Cookie).one() ❷

...b/python2.7/site-packages/sqlalchemy/orm/query.pyc in one(self)
    2480         else:
    2481             raise orm_exc.MultipleResultsFound(
->  2482                 "Multiple rows were found for one()")
    2483
    2484     def scalar(self):

MultipleResultsFound: Multiple rows were found for one() ❸
```

❶ 这行显示了异常类型和错误跟踪。

❷ 这是触发异常的代码行。

❸ 这是需要我们关注的部分。

示例 8-4 显示的是 SQLAlchemy 中 MultipleResultsFound 异常的标准格式。第一行指出异常类型。接下来是异常跟踪，指出异常发生的位置。最后一行提供了重要细节，其中不仅指出了异常类型，还表明了发生异常的原因。本例中，我们要求查询只返回一行，而实际却返回了两行，从而引发了 MultipleResultsFound 异常。



另一个相关异常是 NoResultFound。当我们使用 .one() 方法进行查询时，如果查询没有返回任何结果，就会触发 NoResultFound 异常。

如果想处理这个异常，让程序不会因此而停止执行，并打印出更有用的异常消息，可以使用 Python 的 try/except 块，如示例 8-5 所示。

示例 8-5　处理 MultipleResultsFound 异常

```
from sqlalchemy.orm.exc import MultipleResultsFound ❶
try:
    results = session.query(Cookie).one()
except MultipleResultsFound as error: ❷
    print('We found too many cookies... is that even possible?')
```

❶ 所有 SQLAlchemy ORM 异常都在 sqlalchemy.orm.exc 模块中。

❷ 捕获 MultipleResultsFound 异常，并为其定义别名为 error。

运行示例 8-5 会输出“We found too many cookies... is that even possible?”这样一条信息，同时我们的应用程序会继续正常运行。（我认为不可能找到太多 cookie。）现在，你已经了解了 MultipleResultsFound 异常，并且至少学会了一种处理方法，同时还能够保证程序可以继续执行下去。

8.2.2　DetachedInstanceError

当尝试访问某个实例的某个属性时，如果这个实例需要从数据库加载，但它目前又没有连接到数据库，就会发生 DetachedInstanceError 异常。在讲解这个异常之前，需要先创建要操作的记录。示例 8-6 演示了具体做法。

示例 8-6　创建一个用户和订单

```
cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password')
session.add(cookiemon)
o1 = Order()
o1.user = cookiemon
session.add(o1)

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "Change chocolate chip").one()
line1 = LineItem(order=o1, cookie=cc, quantity=2, extended_cost=1.00)

session.add(line1)
session.commit()
```

现在，我们已经创建好了一个用户，他对应于一个订单和若干行项。到这里，触发 DetachedInstanceError 异常的准备工作就做好了。示例 8-7 演示了如何触发 DetachedInstanceError 异常。

示例 8-7　触发 DetachedInstanceError 异常

```
order = session.query(Order).first()
session.expunge(order)
order.line_items.all()
```

在示例 8-7 中，我们通过查询获取一个 Order 实例。有了这个实例之后，我们又使用 expunge() 把它从会话中分离出来。然后尝试加载 line_items 属性。因为 line_items 是一个关系，所以默认情况下，在你请求数据之前，它是不会加载所有数据的。本例中，我们把实例从会话中分离了，关系缺少会话来执行查询从而加载 line_items 属性，这时就会引发 DetachedInstanceError 异常。

```
DetachedInstanceError                     Traceback (most recent call last)
<ipython-input-35-233bbca5c715> in <module>()
      1 order = session.query(Order).first()
      2 session.expunge(order)
----> 3 order.line_items ❶

site-packages/sqlalchemy/orm/attributes.pyc in __get__(self, instance, owner)
    235             return dict_[self.key]
    236         else:
--> 237             return self.impl.get(instance_state(instance), dict_)
    238
    239

site-packages/sqlalchemy/orm/attributes.pyc in get(self, state, dict_, passive)
    576                     value = callable_(state, passive)
    577                 elif self.callable_:
--> 578                     value = self.callable_(state, passive)
    579                 else:
    580                     value = ATTR_EMPTY

site-packages/sqlalchemy/orm/strategies.pyc in _load_for_state(self, state,
passive)
    499                 "Parent instance %s is not bound to a Session; "
    500                 "lazy load operation of attribute '%s' cannot proceed" %
--> 501                  (orm_util.state_str(state), self.key)
    502             )
    503

DetachedInstanceError: Parent instance <Order at 0x10dc31350> is not bound to
a Session; lazy load operation of attribute 'line_items' cannot proceed ❷
```

❶ 引发异常的代码。

❷ 这是我们要关注的部分。

和示例 8-5 一样，我们阅读异常输出的信息，其中指出，我们试图加载 Order 实例的 line_items 属性，但无法做到这一点，因为该实例并没有绑定到会话。

虽然上面的 DetachedInstanceError 异常是我们有意触发的，但它的行为和其他异常非常相

似，比如 ObjectDeletedError、StaleDataError 和 ConcurrentModificationError。所有这些异常都与实例、会话和数据库之间的信息不一致有关。示例 8-6 中使用 try/except 块处理异常的方法也适用于这些异常，这种处理方法允许代码继续运行。你可以把已分离的实例放入 try 块中检查，并在 except 块中将其添加回会话中。不过，DetachedInstanceError 通常表明在代码的这个地方之前就发生了异常。



关于 ORM 异常的更多细节，请参阅 SQLAlchemy 文档。

到目前为止，我们提到的所有异常（比如 MultipleResultsFound 和 DetachedInstanceError）都和某一条语句的执行失败有关。如果有多个语句，比如在提交或刷新期间执行失败的多个添加或删除语句，就需要使用不同的方式来处理它们。具体来说就是通过手动控制事务来处理它们。下一节将详细讲解有关事务的内容，期间会教你如何使用事务来帮助处理异常和恢复会话。

8.3　事务

事务是一组语句，这组语句被看成一个整体，其执行要么全部成功，要么全部失败。当我们第一次创建会话时，它并没有连接到数据库。当我们对会话执行第一个操作（比如查询）时，它会启动一个连接和一个事务。也就是说，默认情况下，我们不需要手动创建事务。但是，如果我们要处理的异常中，事务的一部分执行成功而另一部分执行失败，或者事务结果引发了异常，这种情况下，我们就必须知道如何手动控制事务。

下面举例说明什么时候需要在当前数据库中这样做。当一位客户向我们订购了 cookie 之后，我们需要把指定的 cookie 发送给客户，然后把它们从库存中删除。但是，如果我们没有足够的 cookie 来完成这笔订单呢？这时，我们需要先查看库存，而不是执行订单。可以使用事务来解决这个问题。

我们需要新开一个 Python shell，创建一些数据表，这里使用第 6 章中的表。但是，我们要向 quantity 列添加 CheckConstraint，这样可以确保它不会小于 0，因为库存中的 cookie 不可能是负数。接下来，重新创建 cookiemon 用户以及 chocolate chip cookie 和 dark chocolate chip cookie 记录，把 chocolate chip cookie 的数量设置为 12 块，把 dark chocolate chip cookie 的数量设置为 1 块。示例 8-8 演示了具体的操作步骤。

示例 8-8　建立事务环境

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')
```

```
Session = sessionmaker(bind=engine)

session = Session()

from datetime import datetime

from sqlalchemy import (Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, CheckConstraint)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, backref


Base = declarative_base()


class Cookie(Base):
    __tablename__ = 'cookies'
    __table_args__ = (CheckConstraint('quantity >= 0', name='quantity_positive'),) ❶

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __init__(self, name, recipe_url=None, sku=None, quantity=0, unit_cost=0.00):
        self.cookie_name = name
        self.cookie_recipe_url = recipe_url
        self.cookie_sku = sku
        self.quantity = quantity
        self.unit_cost = unit_cost

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
               "cookie_recipe_url='{self.cookie_recipe_url}', " \
               "cookie_sku='{self.cookie_sku}', " \
               "quantity={self.quantity}, " \
               "unit_cost={self.unit_cost})".format(self=self)


class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now)

    def __init__(self, username, email_address, phone, password):
        self.username = username
```

```
        self.email_address = email_address
        self.phone = phone
        self.password = password

    def __repr__(self):
        return "User(username='{self.username}', " \
                     "email_address='{self.email_address}', " \
                     "phone='{self.phone}', " \
                     "password='{self.password}')".format(self=self)


class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id'))
    shipped = Column(Boolean(), default=False)

    user = relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
                      "shipped={self.shipped})".format(self=self)


class LineItem(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                               order_by=line_item_id))
    cookie = relationship("Cookie", uselist=False)

    def __repr__(self):
        return "LineItem(order_id={self.order_id}, " \
                         "cookie_id={self.cookie_id}, " \
                         "quantity={self.quantity}, " \
                         "extended_cost={self.extended_cost})".format(
                    self=self)

Base.metadata.create_all(engine)

cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password') ❷
cc = Cookie('chocolate chip', 'http://some.aweso.me/cookie/recipe.html', ❸
            'CC01', 12, 0.50)
dcc = Cookie('dark chocolate chip', ❹
             'http://some.aweso.me/cookie/recipe_dark.html',
             'CC02', 1, 0.75)

session.add(cookiemon)
session.add(cc)
session.add(dcc)
```

❶ 添加数量为正的检查约束（`CheckConstraint`）。

❷ 添加 cookiemon 用户。

❸ 添加 chocolate chip cookie。

❹ 添加 dark chocolate chip cookie。

接下来，为 cookiemon 用户定义两个订单。第一个订单是 9 块 chocolate chip cookie，第二个订单是 2 块 chocolate chip cookie 和 9 块 dark chocolate chip cookie。示例 8-9 显示了添加订单的代码。

示例 8-9　添加订单

```
o1 = Order()
o1.user = cookiemon
session.add(o1)

line1 = LineItem(order=o1, cookie=cc, quantity=9, extended_cost=4.50)


session.add(line1)
session.commit() ❶
o2 = Order()
o2.user = cookiemon
session.add(o2)

line1 = LineItem(order=o2, cookie=cc, quantity=2, extended_cost=1.50)
line2 = LineItem(order=o2, cookie=dcc, quantity=9, extended_cost=6.75)

session.add(line1)
session.add(line2)
session.commit() ❷
```

❶ 添加第一个订单。

❷ 添加第二个订单。

有了这些订单数据，我们就可以学习事务的工作原理了。我们先要定义一个名为 `ship_it` 的函数。这个 `ship_it` 函数接受一个 `order_id` 参数，它会从库存中删除 cookie，并把订单标记为“已发货”。示例 8-10 是 `ship_it` 函数的定义代码。

示例 8-10　定义 `ship_it` 函数

```
def ship_it(order_id):
    order = session.query(Order).get(order_id)
    for li in order.line_items:
        li.cookie.quantity = li.cookie.quantity - li.quantity ❶
        session.add(li.cookie)
    order.shipped = True ❷
    session.add(order)
    session.commit()
    print("shipped order ID: {}".format(order_id))
```

❶ 对于订单中的每个行项，我们都会从相应 cookies 表的 quantity 列中删除行项指定的数量，这样就可以知道还剩下多少 cookie。

❷ 更新订单，将其标记为“已发货”。

每当我们发出一个订单，ship_it 函数就会执行所有必需的操作。我们对第一个订单运行这个函数，然后查询 cookies 表，确保它正确地减少了 cookie 的数量。示例 8-11 展示了具体步骤。

示例 8-11　对第一个订单运行 ship_it 函数

```
ship_it(1) ❶
print(session.query(Cookie.cookie_name, Cookie.quantity).all()) ❷
```

❶ 对第一个 order_id 运行 ship_it 函数。

❷ 查看库存。

运行示例 8-11，会得到如下结果：

```
shipped order ID: 1
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

太好了！ ship_it 函数工作正常。从执行结果中可以看到，库存中没有足够的 cookie 来完成第二个订单。不过，在工作节奏很快的仓库中，这两个订单可能会被同时处理。现在尝试使用 ship_it 函数完成第二个订单，代码如下：

```
ship_it(2)
```

执行结果如下：

```
IntegrityError                            Traceback (most recent call last)
<ipython-input-7-8a7f7805a7f6> in <module>()
 ---> 1 ship_it(2)
<ipython-input-5-c442ae46326c> in ship_it(order_id)
      6     order.shipped = True
      7     session.add(order)
 ---> 8     session.commit()
      9     print("shipped order ID: {}".format(order_id))
...

IntegrityError: (sqlite3.IntegrityError) CHECK constraint failed:
quantity_positive [SQL: u'UPDATE cookies SET quantity=? WHERE
cookies.cookie_id = ?'] [parameters: (-8, 2)]
```

由于我们没有足够的 dark chocolate chip cookie 完成订单，所以最终得到了一个 IntegrityError 错误。这实际上会中断我们当前的会话。如果尝试通过会话发送更多语句（比如查询）来获取 cookie 的列表，将会得到示例 8-12 所示的输出。

示例 8-12：通过错误会话查询

```
print(session.query(Cookie.cookie_name, Cookie.quantity).all())

InvalidRequestError                       Traceback (most recent call last)
<ipython-input-8-90b93364fb2d> in <module>()
 ---> 1 print(session.query(Cookie.cookie_name, Cookie.quantity).all())

...

InvalidRequestError: This Session's transaction has been rolled back due to a
previous exception during flush. To begin a new transaction with this Session,
first issue Session.rollback(). Original exception was: (sqlite3.IntegrityError)
CHECK constraint failed: quantity_positive [SQL: u'UPDATE cookies SET
quantity=? WHERE cookies.cookie_id = ?'] [parameters: ((1, 1), (-8, 2))]
```

去除里面的跟踪细节，可以发现是前面的 `IntegrityError` 引发了 `InvalidRequestError` 异常。要从这个会话状态中恢复，需要手动回滚事务。示例 8-12 中显示的信息可能有点令人困惑，因为里面提到了"This Session's transaction has been rolled back"（这个会话的事务已回滚），但它实际上是让你手动执行回滚。调用会话的 `rollback()` 方法可以把会话恢复过来。每当遇到异常，你都可以调用 `rollback()` 方法进行回滚：

```
session.rollback()
print(session.query(Cookie.cookie_name, Cookie.quantity).all())
```

这段代码会像我们预料的那样正常输出结果：

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

在会话正常工作的情况下，可以使用 `try/except` 块，确保当 `IntegrityError` 异常发生时将打印一条信息并回滚事务，这样应用程序就可以继续正常工作了，如示例 8-13 所示。

示例 8-13　事务型 `ship_it`

```
from sqlalchemy.exc import IntegrityError ❶
def ship_it(order_id):
    order = session.query(Order).get(order_id)
    for li in order.line_items:
        li.cookie.quantity = li.cookie.quantity - li.quantity
        session.add(li.cookie)
    order.shipped = True
    session.add(order)
    try:
       session.commit()
       print("shipped order ID: {}".format(order_id))
    except IntegrityError as error: ❷
       print('ERROR: {!s}'.format(error.orig))
       session.rollback() ❸
```

❶ 导入 `IntegrityError`，以便处理这个异常。

❷ 若无异常发生，提交事务。

❸ 若有异常发生，回滚事务。

下面针对第二个订单重新运行基于事务的 ship_it，如下所示：

```
ship_it(2)

ERROR: CHECK constraint failed: quantity_positive
```

程序不会因为异常而停止，它会打印错误信息，但不会显示异常跟踪细节。让我们使用示例 8-12 中的代码查看一下库存，确保它不会因发送一部分 cookie 而搞乱库存：

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

太棒了！我们的事务型函数没有搞乱库存或者导致应用程序崩溃。我们不需要手动编写大量代码以便回滚到之前的状态。如你所见，事务在这种情况下非常有用，并且使用它可以大大减少手动编码的工作量。

本章我们学习了单行语句和多行语句的异常处理方法。通过在单行语句上添加普通的 try/except 块，可以防止应用程序在碰到数据库语句执行错误时发生崩溃。我们还学习了会话事务如何帮助避免数据库不一致问题，以及当一组语句组执行失败时如何防止程序发生崩溃。下一章中，我们将学习测试代码的方法，以便确保代码的行为符合我们的预期。

第 9 章
使用SQLAlchemy ORM测试

应用程序测试大都包括单元测试和功能测试两部分。但是，在使用 SQLAlchemy 做单元测试时，要正确地模拟一条查询语句或者一个模型可能需要做很多工作。在对数据库做功能测试时，这些工作通常不会带来多大好处。所以，一般人们会为能在单元测试期间轻松模拟的查询创建包装器，或者在单元测试和功能测试中均只针对数据库做测试。我个人比较喜欢使用小巧的包装器函数，但是如果出于某种原因使得这样做没有意义，或者处理的是遗留代码，那我会选择使用模拟方法。

本章讲解如何对数据库做功能测试，以及如何模拟 SQLAlchemy 查询和连接。

9.1 使用测试数据库做测试

在我们的示例程序中，有一个 app.py 文件，其中包含应用程序逻辑；还有一个 db.py 文件，其中包含我们的数据模型和会话。你可以在 CH09/ 文件夹中找到这些文件。

应用程序的结构会对测试方式产生很大影响。在 db.py 文件中，你可以看到我们的数据库是通过 `DataAccessLayer` 类建立的。我们使用这个数据访问类随时初始化数据库引擎和会话。在与 SQLAlchemy 结合使用的 Web 框架中，你会经常看到这种模式。`DataAccessLayer` 类初始化时，`dal` 变量中并不存在引擎和连接。



虽然我们在示例中使用 SQLite 数据库进行测试，但是强烈建议你在生产环境中使用相同的数据库引擎进行测试。如果你在测试中使用了不同的数据库，那么一些可以在 SQLite 中使用的约束和语句可能在其他数据库上无法正常工作，比如 PostgreSQL。

示例 9-1 显示的是 db.py 文件中的一个片段——DataAccessLayer 类。

示例 9-1　DataAccessLayer 类

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class DataAccessLayer:

    def __init__(self): ❶
        self.engine = None
        self.conn_string = 'some conn string'

    def connect(self): ❷
        self.engine = create_engine(self.conn_string)
        Base.metadata.create_all(self.engine)
        self.Session = sessionmaker(bind=self.engine)


dal = DataAccessLayer() ❸
```

❶ __init__ 提供了一种像工厂一样使用指定连接字符串初始化连接的方法。

❷ connect 方法创建 Base 类中的所有表，它使用 sessionmaker 创建会话，供我们在程序中使用。

❸ 这行代码创建了 DataAccessLayer 类的一个实例，你可以在应用程序中导入它。

除了 DataAccessLayer 之外，我们还在 db.py 文件中定义了所有数据模型。这些模型和我们在第 8 章中使用的模型是一样的，我把它们都放在了这里，以方便在应用程序中使用。下面是 db.py 文件中的所有模型：

```
from datetime import datetime

from sqlalchemy import (Column, Integer, Numeric, String, DateTime, ForeignKey,
                        Boolean, create_engine)
from sqlalchemy.orm import relationship, backref, sessionmaker

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
            "cookie_recipe_url='{self.cookie_recipe_url}', " \
```

```
            "cookie_sku='{self.cookie_sku}', " \
            "quantity={self.quantity}, " \
            "unit_cost={self.unit_cost})".format(self=self)


class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now)

    def __repr__(self):
        return "User(username='{self.username}', " \
            "email_address='{self.email_address}', " \
            "phone='{self.phone}', " \
            "password='{self.password}')".format(self=self)


class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id'))
    shipped = Column(Boolean(), default=False)

    user = relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
            "shipped={self.shipped})".format(self=self)


class LineItem(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))
    cookie = relationship("Cookie", uselist=False)

    def __repr__(self):
        return "LineItem(order_id={self.order_id}, " \
            "cookie_id={self.cookie_id}, " \
            "quantity={self.quantity}, " \
            "extended_cost={self.extended_cost})".format(
                self=self)
```

准备好数据模型和访问类之后，接下来看看待测试的代码。我们将为第 7 章中构建的 get_orders_by_customer 函数编写测试，你可以在 app.py 文件找到它，如示例 9-2 所示。

示例 9-2　待测试的 app.py

```
from db import dal, Cookie, LineItem, Order, User, DataAccessLayer ❶


def get_orders_by_customer(cust_name, shipped=None, details=False):
    query = dal.session.query(Order.order_id, User.username, User.phone) ❷
    query = query.join(User)
    if details:
        query = query.add_columns(Cookie.cookie_name, LineItem.quantity,
                                  LineItem.extended_cost)
        query = query.join(LineItem).join(Cookie)
    if shipped is not None:
        query = query.filter(Order.shipped == shipped)
    results = query.filter(User.username == cust_name).all()
    return results
```

❶ 这是我们的 DataAccessLayer 实例（来自于 db.py 文件）和要在程序代码中使用的数据模型。

❷ 由于 session 是 dal 对象的一个属性，所以我们需要通过 dal 来访问它。

下面看看 get_orders_by_customer 函数的所有使用方法。在这个练习中，假设我们已经对函数的输入进行了验证，它们的类型都是正确的。不过，在你的测试中，明智的做法是确保你使用的数据有能正常工作的，也有可能会引起错误的。下面是函数的参数及其可能的取值。

- cust_name 可以是空的、包含有效客户名的字符串，或者是不包含有效客户名的字符串。
- shipped 可以是 None、True 或 False。
- details 可以是 Ture 或 False。

如果想测试所有可能的组合，那么需要 12（即 3 * 3 * 2）个测试来测试这个函数。



请注意，不要测试那些属于 SQLAlchemy 基本功能的代码，因为 SQLAlchemy 本身已经做了大量良好的测试。比如，我们不会测试简单的 insert、select、delete 或 update 语句，因为这些语句已经在 SQLAlchemy 项目中测试过了。我们应该测试代码操作的内容，这些内容可能会影响 SQLAlchemy 语句的运行方式或返回的结果。

对于这个测试示例，我们会使用内置的 unittest 模块。如果你不熟悉这个模块，也不必担心，我会给大家讲讲重点内容。首先，我们需要编写测试类，并对 dal 的连接进行初始化。为此，新建一个名为 test_app.py 的文件，代码如示例 9-3 所示。

示例 9-3　创建测试

```
import unittest

from db import dal


class TestApp(unittest.TestCase): ❶

    @classmethod
    def setUpClass(cls): ❷
        dal.conn_string = 'sqlite:///:memory:' ❸
        dal.connect()
```

❶ `unittest` 需要的测试类继承自 `unittest.TestCase`。

❷ 测试开始之前，`setUpClass` 方法要先执行一次。

❸ 为测试设置连接字符串，以便连接到内存数据库。

在数据库连接好之后，就可以动手编写一些测试了。我们会把这些测试以函数的形式添加到 `TestApp` 类中，如示例 9-4 所示。

示例 9-4　针对空用户名的前 6 个测试

```
def test_orders_by_customer_blank(self): ❶
    results = get_orders_by_customer('')
    self.assertEqual(results, []) ❷

def test_orders_by_customer_blank_shipped(self):
    results = get_orders_by_customer('', True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped(self):
    results = get_orders_by_customer('', False)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_details(self):
    results = get_orders_by_customer('', details=True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_shipped_details(self):
    results = get_orders_by_customer('', True, True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped_details(self):
    results = get_orders_by_customer('', False, True)
    self.assertEqual(results, [])
```

❶ `unittest` 要求每个测试都以 `test` 打头。

❷ `unittest` 使用 `assertEqual` 方法来验证结果是否符合你的预期。由于没有找到用户，所以你应该会得到一个空列表。

现在，把测试文件保存为 test_app.py，然后使用如下命令运行单元测试：

```
# python -m unittest test_app
......

Ran 6 tests in 0.018s
```



运行单元测试时，你可能会得到一个有关 SQLite 和 decimal 类型的警告。忽略它就好，因为这在我们的例子中是正常的。之所以会发出警告，是因为 SQLite 中没有真正的 decimal 类型，而 SQLAlchemy 希望你知道，从 SQLite 的 float 类型转换而来可能有些奇怪。认真阅读这些消息是明智之举，因为在生产代码中，这些信息通常能为你指出正确的做事方法。这里我们特意触发这个警告，以便让你看看它的样子。

接下来需要加载一些数据，并确保我们的测试仍然有效。同样，我们会重用第 7 章中的例子，插入相同的用户、订单和行项。不过，这一次我们会把数据插入工作放入一个名为 `prep_db` 的函数中，这样就可以在测试之前通过调用 `prep_db` 这个函数来插入数据了。为了简单起见，我把这个函数放入 db.py 文件中（参见示例 9-5）。但是，在真实情况下，我们通常会把它放到测试装置或实用程序文件中。

示例 9-5　插入一些测试数据

```
def prep_db(session):
    c1 = Cookie(cookie_name='dark chocolate chip',
                cookie_recipe_url='http://some.aweso.me/cookie/dark_cc.html',
                cookie_sku='CC02',
                quantity=1,
                unit_cost=0.75)
    c2 = Cookie(cookie_name='peanut butter',
                cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
                cookie_sku='PB01',
                quantity=24,
                unit_cost=0.25)
    c3 = Cookie(cookie_name='oatmeal raisin',
                cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
                cookie_sku='EWW01',
                quantity=100,
                unit_cost=1.00)
    session.bulk_save_objects([c1, c2, c3])
    session.commit()

    cookiemon = User(username='cookiemon',
                     email_address='mon@cookie.com',
                     phone='111-111-1111',
                     password='password')
    cakeeater = User(username='cakeeater',
                     email_address='cakeeater@cake.com',
                     phone='222-222-2222',
                     password='password')
```

```
        pieperson = User(username='pieperson',
                         email_address='person@pie.com',
                         phone='333-333-3333',
                         password='password')
        session.add(cookiemon)
        session.add(cakeeater)
        session.add(pieperson)
        session.commit()

        o1 = Order()
        o1.user = cookiemon
        session.add(o1)

    line1 = LineItem(cookie=c1, quantity=2, extended_cost=1.00)

    line2 = LineItem(cookie=c3, quantity=12, extended_cost=3.00)

    o1.line_items.append(line1)
    o1.line_items.append(line2)
    session.commit()

    o2 = Order()
    o2.user = cakeeater

    line1 = LineItem(cookie=c1, quantity=24, extended_cost=12.00)
    line2 = LineItem(cookie=c3, quantity=6, extended_cost=6.00)

    o2.line_items.append(line1)
    o2.line_items.append(line2)

    session.add(o2)
    session.commit()
```

到这里，prep_db 函数就创建好了，现在可以在 test_app.py 文件的 setUpClass 方法中调用它，把数据加载到数据库，再运行测试。setUpClass 方法的代码如下：

```
@classmethod
def setUpClass(cls):
    dal.conn_string = 'sqlite:///:memory:'
    dal.connect()
    dal.session = dal.Session()
    prep_db(dal.session)
    dal.session.close()
```

每次测试之前，还需要创建一个新会话，并在每次测试之后回滚会话期间所做的所有更改。为此，可以添加一个 setup 方法，它在每个测试之前自动运行；还要添加一个 tearDown 方法，它在每个测试之后自动运行，如下所示：

```
def setUp(self): ❶
    dal.session = dal.Session()

def tearDown(self): ❷
```

```
        dal.session.rollback()
        dal.session.close()
```

❶ 每次测试之前，新建一个会话。

❷ 每次测试后，回滚测试期间所做的更改，并清除会话。

我们还将以属性的形式把期望的结果添加到 TestApp 类，如下所示：

```
    cookie_orders = [(1, u'cookiemon', u'111-111-1111')]
    cookie_details = [
        (1, u'cookiemon', u'111-111-1111',
            u'dark chocolate chip', 2, Decimal('1.00')),
        (1, u'cookiemon', u'111-111-1111',
            u'oatmeal raisin', 12, Decimal('3.00'))]
```

接下来就可以使用这些测试数据来确保我们的函数对给定的有效用户名执行正确的操作。把测试以函数的形式添加到 TestApp 类中，如示例 9-6 所示。

示例 9-6　测试有效用户

```
    def test_orders_by_customer(self):
        results = get_orders_by_customer('cookiemon')
        self.assertEqual(results, self.cookie_orders)

    def test_orders_by_customer_shipped_only(self):
        results = get_orders_by_customer('cookiemon', True)
        self.assertEqual(results, [])

    def test_orders_by_customer_unshipped_only(self):
        results = get_orders_by_customer('cookiemon', False)
        self.assertEqual(results, self.cookie_orders)

    def test_orders_by_customer_with_details(self):
        results = get_orders_by_customer('cookiemon', details=True)
        self.assertEqual(results, self.cookie_details)

    def test_orders_by_customer_shipped_only_with_details(self):
        results = get_orders_by_customer('cookiemon', True, True)
        self.assertEqual(results, [])

    def test_orders_by_customer_unshipped_only_details(self):
        results = get_orders_by_customer('cookiemon', False, True)
        self.assertEqual(results, self.cookie_details)
```

使用示例 9-6 中的测试作为指导，你能完成针对不同用户（比如 cakeeater）的测试吗？能对系统中不存在的用户名进行测试吗？如果用户名是整数而不是字符串，结果会怎样？请将你的测试和示例代码中的测试进行比较，看看你的测试是否与本书中使用的测试相似。

到这里，我们已经学习了如何在功能测试中使用 SQLAlchemy 来判断一个函数在给定数据集上的行为是否与预期一致。我们还了解了如何设置 unittest 文件，以及如何准备要在测试中使用的数据库。接下来，我们在不访问数据库的情况下进行测试。

9.2 使用mock

当在你的测试环境中创建测试数据库没有意义或者根本不可行时，mock 技术会派上大用场。如果你有大量逻辑需要对查询结果进行操作，那么模拟 SQLAlchemy 代码返回你想要的值会很有用，这样你就可以只测试周边逻辑。通常，在模拟查询的某个部分时，我仍然会创建内存数据库，但是我不会向其中加载任何数据，而会模拟数据库连接本身。这样我就可以控制执行和获取方法返回的内容。我们将在本节中学习如何做到这一点。

为了学习如何在测试中使用 mock，我们会为一个有效用户做一次测试。我们将使用强大的 mock 库来控制连接返回的内容。在 Python 3 中，mock 是 `unittest` 模块的一部分。但是，如果你使用的是 Python 2，那么需要先使用 pip 来安装 mock 库，以便获得最新的功能。为此，可以使用以下命令：

```
pip install mock
```

至此，我们已经安装好 mock 库，接下来就可以在测试中使用它了。mock 有一个 `patch()` 函数，它允许我们使用一个可以在测试中控制的 `MagicMock` 替换 Python 文件中给定的对象。`MagicMock` 是一个特殊的 Python 对象，它可以跟踪自身的使用方式，并允许我们根据其使用方式来定义它的行为。

首先，需要导入 mock 库。在 Python 2 中，使用如下代码：

```
import mock
```

在 Python 3 中，使用如下代码：

```
from unittest import mock
```

导入 mock 库之后，我们将把 `patch` 方法用作装饰器来替换 `dal` 对象的会话。装饰器函数用来包装另外一个函数，并可改变被包装函数的行为。因为 `dal` 对象是通过名称导入到 app.py 文件的，所以我们需要在 `app` 模块中包装它，然后将其作为参数传入测试函数。现在已经有了一个 mock 对象，接下来就可以为 `execute` 方法设置一个返回值了，本例中应该什么也不返回，但要接上 `fetchall` 方法，它的返回值是我们想要测试的数据。示例 9-7 给出了使用 mock 替换 `dal` 对象所需的代码。

示例 9-7　模拟连接测试

```
import unittest
from decimal import Decimal

import mock

from app import get_orders_by_customer

class TestApp(unittest.TestCase):
```

```
    cookie_orders = [(1, u'cookiemon', u'111-111-1111')]
    cookie_details = [
        (1, u'cookiemon', u'111-111-1111',
            u'dark chocolate chip', 2, Decimal('1.00')),
        (1, u'cookiemon', u'111-111-1111',
            u'oatmeal raisin', 12, Decimal('3.00'))]

    @mock.patch('app.dal.session') ❶
    def test_orders_by_customer(self, mock_dal): ❷
        mock_dal.query.return_value.join.return_value.filter.return_value. \
            all.return_value = self.cookie_orders ❸
        results = get_orders_by_customer('cookiemon') ❹
        self.assertEqual(results, self.cookie_orders)
```

❶ 使用 mock 替换 `app` 模块中的 `dal.session`。

❷ mock 以 `mock_dal` 的形式传入测试函数。

❸ 把 `execute` 方法的返回值设置成 `all` 方法的返回值，并将 `all` 方法的返回值设置为 `self.cookie_order`。

❹ 调用测试函数，模拟 `dal.connection`，并返回上一步设置的返回值。

你可以看到，使用复杂的查询（如示例 9-7 所示）尝试模拟完整的查询或连接时，可能很快就会变得非常乏味。但不要因此而不这样做，这对于发现未知 bug 非常有用。

我希望大家把其余的测试作为练习，自己使用内存数据库和模拟测试类型来完成它们。此外，建议你把查询和连接都模拟一下，这样你会对整个模拟过程更熟悉。

到这里，你应该已经熟悉如何测试包含 SQLAlchemy ORM 函数和模型的函数。你还学会了如何把数据预填充到测试数据库中，以便在测试中使用。最后，还学会了如何模拟查询对象和连接对象。本章举的例子比较简单，我们将在第 14 章中深入学习测试，届时会涉及 Flask、Pyramid 和 pytest。

接下来，我们将学习在 Python 中如何在不重建所有模式的情形下使用 SQLAlchemy 处理现有数据库，这会用到反射和自动映射。

第10章

使用SQLAlchemy ORM和自动映射进行反射

在第 5 章我们学过，反射允许你使用现有数据库填充 SQLAlchemy 对象。反射可以应用到表、视图、索引和外键上。但是，如果你想把数据库模式反射到 ORM 风格的类中呢？幸运的是，SQLAlchemy 扩展模块中的 automap（自动映射）允许你这样做。

基于 automap 的反射是一个非常有用的工具。然而，在 SQLAlchemy 1.0 版本中，我们不能反射 CheckConstraints、注释和触发器。你也不能反射客户端的默认值，以及序列与列之间的关联。但是，我们可以使用第 6 章讲的方法手动添加它们。

如同第 5 章，我们使用 Chinook 数据库做测试。这里，我们用的是 Chinook 数据库的 SQLite 版本，你可以在本书示例代码的 CH10 文件夹中找到它。这个文件夹中还包含一张数据库模式图像，通过它你可以清晰地了解本章中使用的模式。

10.1 使用自动映射反射数据库

为了反射数据库，我们会使用 automap_base，而不是之前一直使用的 ORM declarative_base。先创建一个 Base 对象，如示例 10-1 所示。

示例 10-1 使用 automap_base 创建 Base 对象

```
from sqlalchemy.ext.automap import automap_base ❶

Base = automap_base() ❷
```

❶ 从 automap 扩展导入 `automap_base`。

❷ 初始化 `Base` 对象。

接下来需要将一个引擎连接到想反射的数据库。示例 10-2 演示了如何连接到 Chinook 数据库。

示例 10-2　为 Chinook 数据库初始化引擎

```
from sqlalchemy import create_engine

engine = create_engine('sqlite:///Chinook_Sqlite.sqlite') ❶
```

❶ 连接字符串假设当前文件和示例数据库在同一目录下。

当 `Base` 和引擎创建好之后，我们就做好了反射数据库的准备工作。调用示例 10-1 中创建的 `Base` 对象的 `prepare` 方法，将扫描我们刚刚创建的引擎上的所有可用内容，并反射它所能反射的所有内容。下面是使用 `Base` 对象反射数据库的代码：

```
Base.prepare(engine, reflect=True)
```

只需要这一行代码，就完成了整个数据库的映射工作。这个反射已经为每个表创建了 ORM 对象。可以在 `Base` 的 `class` 属性下访问这些表。要打印出这些对象，只需运行如下代码：

```
Base.classes.keys()
```

执行代码，会得到如下结果：

```
['Album',
 'Customer',
 'Playlist',
 'Artist',
 'Track',
 'Employee',
 'MediaType',
 'InvoiceLine',
 'Invoice',
 'Genre']
```

接下来创建两个引用 `Artist` 和 `Album` 表的对象：

```
Artist = Base.classes.Artist
Album = Base.classes.Album
```

第一行代码创建了一个到 `Artist` ORM 对象的引用，第二行代码创建了一个到 `Album` ORM 对象的引用。可以像使用第 7 章手动定义的 ORM 对象一样使用 `Artist` 对象。示例 10-3 演示了如何使用对象执行简单的查询，以获取表中的前 10 条记录。

示例 10-3　使用 `Artist` 表

```
from sqlalchemy.orm import Session

session = Session(engine)
for artist in session.query(Artist).limit(10):
    print(artist.ArtistId, artist.Name)
```

运行示例 10-3，会得到如下结果：

```
(1, u'AC/DC')
(2, u'Accept')
(3, u'Aerosmith')
(4, u'Alanis Morissette')
(5, u'Alice In Chains')
(6, u'Ant\xf4nio Carlos Jobim')
(7, u'Apocalyptica')
(8, u'Audioslave')
(9, u'BackBeat')
(10, u'Billy Cobham')
```

现在，我们已经学会如何反射数据库，以及如何将其映射到对象。接下来学习使用自动映射反射关系的方法。

10.2　反射关系

自动映射可以自动反射和建立多对一、一对多和多对多的关系。接下来看看 `Album` 和 `Artist` 表之间的关系是如何建立的。当 automap 创建一个关系时，它会在对象上创建一个 `<related_object>_collection` 属性，请看示例 10-4 中的 `Artist` 对象。

示例 10-4　使用 `Artist` 和 `Album` 之间的关系打印相关数据

```
artist = session.query(Artist).first()
for album in artist.album_collection:
    print('{} - {}'.format(artist.Name, album.Title))
```

运行上面的代码，会得到如下结果：

```
AC/DC - For Those About To Rock We Salute You
AC/DC - Let There Be Rock
```

你还可以配置自动映射，并修改它的某些行为，这样可以让创建的类更符合特定要求。不过，这些内容远远超出了本书的讨论范围。你可以在 SQLAlchemy 文档中了解更多这方面的内容。

本章讲解了 SQLAlchemy ORM 的重要内容。希望你通过本章的学习，认识到了 SQLAlchemy 的反射功能是多么地强大。本书为你学习 ORM 打下了坚实的基础，但要学的内容还有很多。要想获取更多内容，请参阅 SQLAlchemy 文档。

在下一部分，我们将学习如何使用 Alembic 来管理数据库迁移，以便在不破坏和重新创建数据库的情况下更改数据库模式。

第三部分

Alembic

Alembic 是一个处理数据库更改的工具，它利用 SQLAlchemy 来执行迁移。当我们使用元数据的 `create_all` 方法时，SQLAlchemy 只会创建缺失的表，所以它不会更新数据库表以反映我们对列所做的更改，而且它也不会删除我们已经在代码中删除的表。Alembic 提供了一种添加 / 删除表、更改列名和添加新约束的方法。由于 Alembic 使用 SQLAlchemy 做迁移，所以可以在大量后端数据库上使用。

第 11 章

Alembic入门

Alembic 提供了一种创建和执行迁移的程序化方法，让我们可以随着程序的演进灵活地处理需要对数据库做的更改。例如，我们可以向表中添加列，或者从模型中删除属性，还可以添加全新的模型，或者把现有模型分解为多个模型。Alembic 为我们提供了一种利用 SQLAlchemy 的强大功能来做这些更改的方法。

开始之前，需要先安装 Alembic。可以使用如下代码安装 Alembic：

```
pip install alembic
```

安装好 Alembic 之后，还需要创建迁移环境。

11.1 创建迁移环境

为了创建迁移环境，首先创建一个名为 CH11 的文件夹，并进入这个目录。而后，运行 `alembic init alembic` 命令，在 alembic/ 目录中创建迁移环境。人们通常会选择在 migrations/ 目录中创建迁移环境，为此可以使用 `alembic init migrations` 命令。你也可以随意为目录起一个自己喜欢的名字，但我建议你起一个与众不同的名字，并且一定不要和某个模块的名字重复。这个初始化过程会创建迁移环境并创建一个 alembic.ini 文件，这个文件里面包含配置选项。现在，查看我们的目录，会看到以下结构：

```
.
├── alembic
│   ├── README
│   ├── env.py
│   ├── script.py.mako
│   └── versions
└── alembic.ini
```

在新创建的迁移环境中，我们会看到一个 env.py 文件、一个 script.py.mako 模板文件，以及一个 versions/ 目录。其中，versions/ 目录用来保存迁移脚本。Alembic 会使用 env.py 文件定义和实例化 SQLAlchemy 引擎，然后连接到这个引擎，启动一个事务，并且当你运行 Alembic 命令时，它会正确地调用迁移引擎。script.py.mako 模板用来创建迁移，它定义了迁移的基本结构。

创建好迁移环境之后，要对它进行配置，以使其和我们的应用程序协同工作。

11.2 配置迁移环境

首先调整 alembic.ini 和 env.py 文件中的设置，这样 Alembic 才能与我们的数据库和应用程序一起工作。让我们从 alembic.ini 文件开始，更改其中的 `sqlalchemy.url` 配置项，将其修改为我们的数据库连接字符串。这里，我们想让它连接到当前目录下一个名为 alembictest.db 的 SQLite 文件。为此，修改 `sqlalchemy.url` 行，如下所示：

```
sqlalchemy.url = sqlite:///alembictest.db
```

在所有 Alembic 示例中，我们会使用 ORM 的声明式风格在 app/db.py 文件中创建所有代码。首先，创建一个 app/ 目录，并在其中创建一个空的 __init__.py 文件，让 `app` 成为一个模块。

然后，在 app/db.py 中添加如下代码，对 SQLAlchemy 进行设置，使其使用的数据库和 alembic.ini 文件中（对 Alembic）配置的一样，并且定义一个基类。

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///alembictest.db')

Base = declarative_base()
```

接下来更改 env.py 文件，让其指向我们的 `metadata`，`metadata` 是我们在 app/db.py 中创建的 `Base` 实例的一个属性。env.py 会使用 `metadata` 将在数据库中找到的数据与 SQLAlchemy 中定义的模型进行比较。首先，要把当前目录添加到 Python 用来定位模块的路径中，这样它就可以找到我们的 `app` 模块了：

```
import os
import sys

sys.path.append(os.getcwd()) ❶
```

❶ 将当前工作目录（CH11）添加到 sys.path 中，Python 会在 sys.path 中搜索模块。

最后，修改 env.py 文件中的 `target_metadata` 一行，将其改为 app/db.py 文件中的 `metadata` 对象，如下所示：

```
from app.db import Base ❶
target_metadata = Base.metadata ❷
```

❶ 导入 `Base` 实例。

❷ 让 Alembic 把 `Base.metadata` 作为目标。

完成这一步之后，我们就设置好 Alembic 环境了，此时它可以使用应用程序的数据库和元数据。我们还构建好了一个应用程序框架，我们会在其中定义数据模型，相关内容在第 12 章讲解。

本章中，我们学习了如何创建迁移环境，以及如何配置它，以使其共享应用程序的数据库和元数据。在下一章中，我们会学习创建迁移的两种方式：一种是手动方式，另一种是使用自动生成功能。

第 12 章 创建迁移

在第 11 章中，我们初始化并配置好了 Alembic 迁移环境。接下来就可以向应用程序添加数据类，以及创建迁移并把它们添加到数据库中了。本章，我们会学习如何使用自动生成方式添加表，以及如何用手动迁移来完成自动生成方式无法做到的事情。让我们从一个空迁移开始做起，空迁移可以为我们的迁移提供一个干净的起点。

12.1　创建基础空迁移

为了创建基础空迁移，首先要确保你当前在 CH12 文件夹下，而后使用如下命令创建空迁移：

```
# alembic revision -m "Empty Init" ❶
   Generating ch12/alembic/versions/8a8a9d067_empty_init.py ... done
```

❶ 运行 `alembic revision` 命令，并向迁移添加信息 (`-m`) `"Empty Init"`。

这会在 alembic/versions/ 子文件夹下创建一个迁移文件。文件名总是以代表修订 ID 的“#”开头，然后是你提供的信息。下面看看文件内部：

```
"""Empty Init ❶

Revision ID: 8a8a9d067
Revises:
Create Date: 2015-09-13 20:10:05.486995

"""

# revision identifiers, used by Alembic.
```

```
revision = '8a8a9d067' ❷
down_revision = None ❸
branch_labels = None ❹
depends_on = None ❺

from alembic import op
import sqlalchemy as sa


def upgrade():
    pass


def downgrade():
    pass
```

❶ 我们指定的迁移信息。

❷ 修订 ID。

❸ 用来确定如何降级的前一次修订。

❹ 和这次迁移相关的分支。

❺ 本次迁移依赖的迁移。

文件头部分包含我们提供的信息（如果有的话）、修订 ID 以及创建日期和时间。接下来是标识符部分，它指出迁移的性质，以及降级或依赖的细节，或者和本次迁移相关的分支。一般来说，如果你在代码的一个分支中修改数据类，那你可能还需要为 Alembic 迁移建立分支。

接下来是 `upgrade` 方法，里面包含执行迁移时用于改动数据库的所有 Python 代码。最后是 `downgrade` 方法，里面的代码用来撤销本次迁移，并把数据库恢复到先前的迁移步骤。

由于目前没有任何数据类，也没有做任何更改，所以 `upgrade` 和 `downgrade` 方法都是空的。因此，运行这个迁移不会产生任何影响，但它为我们的迁移链提供了一个很好的基础。为了完成从当前数据库状态到最高 Alembic 迁移的所有迁移，要执行以下命令：

```
# alembic upgrade head ❶
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.runtime.migration] Running upgrade -> 8a8a9d067, Empty Init
```

❶ 将数据库升级到最新版本。

前面的输出会把数据库最新修订以前的每次修订都列出来。在本例中，只有一个“Empty Init”迁移，它最后运行。

在此基础之上，就可以把用户数据类添加到应用程序了。添加好数据类之后，就可以创建一个自动生成的迁移，并使用它在数据库中创建关联表。

12.2 自动生成迁移

接下来，让我们把用户数据类添加到 app/db.py 中。这里添加的是 `Cookie` 类，它和前面讲 ORM 时使用的类是一样的。首先从 `sqlalchemy` 导入编写 `Cookie` 类时要使用的 `Column` 和列类型，然后再添加 `Cookie` 类，如示例 12-1 所示。

示例 12-1　修改 app/db.py

```
from sqlalchemy import create_engine, Column, Integer, Numeric, String

from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///alembictest.db')

Base = declarative_base()


class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))
```

添加好 `Cookie` 类之后，接下来创建迁移，把这个表添加到数据库。这个迁移非常简单，可以使用 Alembic 自动生成它（见示例 12-2）。

示例 12-2　自动生成迁移

```
# alembic revision --autogenerate -m "Added Cookie model" ❶
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'cookies'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_cookies_cookie_name'
      on '['cookie_name']'
  Generating ch12/alembic/versions/34044511331_added_cookie_model.py ... done
```

❶ 使用“Added Cookie model”这个信息自动生成一个迁移。

因此，当运行自动生成命令时，Alembic 首先会检查 SQLAlchemy`Base` 的元数据，而后将其与当前数据库状态进行比较。在示例 12-2 中，它检测到我们添加了一个名为 `cookies` 的表（基于 `__tablename__`）和该表 `cookie_name` 字段上的索引。它把这些差异标记为更改，并添加逻辑以便在迁移文件中创建它们。下面看一下它创建的迁移文件，如示例 12-3 所示。

示例 12-3　cookies 迁移文件

```
"""Added Cookie model

Revision ID: 34044511331
Revises: 8a8a9d067
Create Date: 2015-09-14 20:37:25.924031

"""

# revision identifiers, used by Alembic.
revision = '34044511331'
down_revision = '8a8a9d067'
branch_labels = None
depends_on = None

from alembic import op
import sqlalchemy as sa

def upgrade():
    ### commands auto generated by Alembic - please adjust! ###
    op.create_table('cookies', ❶
    sa.Column('cookie_id', sa.Integer(), nullable=False),
    sa.Column('cookie_name', sa.String(length=50), nullable=True),
    sa.Column('cookie_recipe_url', sa.String(length=255), nullable=True),
    sa.Column('cookie_sku', sa.String(length=55), nullable=True),
    sa.Column('quantity', sa.Integer(), nullable=True),
    sa.Column('unit_cost', sa.Numeric(precision=12, scale=2), nullable=True),
    sa.PrimaryKeyConstraint('cookie_id') ❷
    )
    op.create_index(op.f('ix_cookies_cookie_name'), 'cookies', ['cookie_name'],
                    unique=False) ❸
    ### end Alembic commands ###

def downgrade():
### commands auto generated by Alembic - please adjust! ###
op.drop_index(op.f('ix_cookies_cookie_name'), table_name='cookies') ❹
op.drop_table('cookies') ❺
### end Alembic commands ###
```

❶ 使用 Alembic 的 `create_table` 方法添加 `cookies` 表。

❷ 在 `cookie_id` 列上添加主键。

❸ 使用 Alembic 的 `create_index` 方法，在 `cookie_name` 列上添加索引。

❹ 使用 Alembic 的 `drop_index` 方法删除 `cookie_name` 索引。

❺ 删除 `cookies` 表。

在示例 12-3 的 `upgrade` 方法中，`create_table` 方法的语法与第 1 章讲 SQLAlchemy Core 时使用的 `Table` 构造函数是一样的：先是表名，而后是表的列和约束。`create_index` 方法也和第 1 章中使用的 `Index` 构造函数一样。

最后，示例 12-3 中的 downgrade 方法按正确的顺序使用了 Alembic 的 drop_index 和 drop_table 方法，确保索引和表都删除了。

参考前面运行空白迁移的命令，运行这个迁移，如下：

```
# alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 8a8a9d067 -> 34044511331,
Added Cookie model
```

运行之后，查看一下数据库，看看更改是否生效了：

```
# sqlite3 alembictest.db ❶
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .tables ❷
alembic_version cookies
sqlite> .schema cookies ❸
CREATE TABLE cookies (
        cookie_id INTEGER NOT NULL,
        cookie_name VARCHAR(50),
        cookie_recipe_url VARCHAR(255),
        cookie_sku VARCHAR(55),
        quantity INTEGER,
        unit_cost NUMERIC(12, 2),
        PRIMARY KEY (cookie_id)
);
CREATE INDEX ix_cookies_cookie_name ON cookies (cookie_name);
```

❶ 使用 sqlite3 命令访问数据库。

❷ 列出数据库中的表。

❸ 打印 cookies 表的结构。

太好了！我们的迁移完美地创建出了表和索引。不过，Alembic 的自动生成功能本身有一些局限性。表 12-1 列出了自动生成功能可以检测到的一些常见模式变化，而表 12-2 则列出了自动生成功能不能检测到或检测不准的一些模式变化。

表12-1：自动生成功能可以检测到的模式变化

模式元素	更　改
表	添加和删除
列	添加、删除、列的“允许空值”状态的变化
索引	索引的基本变化、显式命名的唯一性约束、支持自动生成索引和唯一性约束
键	基本的重命名

表12-2：自动生成功能无法检测到的模式变化

模式元素	更　　改
表	名称变化
列	名称变化
约束	无明确名称的约束
类型	ENUM 这类数据库后端不直接支持的类型

除了自动生成功能支持和不能支持的功能之外，还有一些可选功能需要做特殊的配置或定制代码才能实现。例如，对列类型的更改或对服务器默认值的更改。如果想了解更多有关自动生成的功能和局限性的内容，可以阅读 Alembic autogenerate 文档。

12.3 手动创建迁移

Alembic 不能检测表名的变化，所以下面了解一下如何在不使用自动生成功能的情况下检测到表名的变化，并将 `cookies` 表更改为 `new_cookies` 表。我们需要先更改 app/db.py 的 `Cookie` 类中的 `__tablename__`，如下所示：

```
class Cookie(Base):
    __tablename__ = 'new_cookies'
```

接下来需要新建一个迁移，我们可以使用“Renaming cookies to new_cookies”编辑它：

```
# alembic revision -m "Renaming cookies to new_cookies"
  Generating ch12/alembic/versions/2e6a6cc63e9_renaming_cookies_to_new_cookies.py
... done
```

迁移建好之后，还要编辑迁移文件，并向 `upgrade` 和 `downgrade` 方法添加重命名操作，如下所示：

```
def upgrade():
    op.rename_table('cookies', 'new_cookies')

def downgrade():
    op.rename_table('new_cookies', 'cookies')
```

接下来，使用 `alembicupgrade` 命令运行迁移：

```
# alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 34044511331 -> 2e6a6cc63e9,
      Renaming cookies to new_cookies
```

这会把数据库中的表重命名为 `new_cookies`。然后，使用 `sqlite3` 命令，查看一下更改是否生效：

```
± sqlite3 alembictest.db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .tables
alembic_version new_cookies
```

和我们想的一样，cookies 表不复存在，因为它已经被 new_cookie 表替代。除了 rename_table 之外，Alembic 中还有许多操作可以帮助你更改数据库，如表 12-3 所示。

表12-3：Alembic操作

操　作	用　途
add_column	添加新列
alter_column	更改列类型、服务器默认值或名称
create_check_constraint	添加新的检查约束
create_foreign_key	添加新的外键
create_index	添加新索引
create_primary_key	添加新主键
create_table	添加新表
create_unique_constraint	添加新的唯一性约束
drop_column	删除列
drop_constraint	删除约束
drop_index	删除索引
drop_table	删除表
execute	运行原始 SQL 语句
rename_table	重命名表



虽然 Alembic 支持表 12-3 中的所有操作，但并不是每个后端数据库都支持它们。比如，alter_column 无法在 SQLite 数据库上工作，因为 SQLite 不支持以任何方式修改列。SQLite 也不支持删除列。

在本章中，我们学习了实现自动迁移和手动迁移的方法，以便以一种可重复的方式对数据库进行更改。我们还学习了如何使用 alembic upgrade 命令让迁移生效。请记住，除了使用 Alembic 操作之外，还可以在迁移中使用 Python 代码来帮助我们实现目标。在下一章中，我们将学习如何进行降级和进一步控制 Alembic。

第 13 章
控制Alembic

在上一章中，我们学习了如何创建和应用迁移，这一章将讨论如何进一步控制 Alembic。我们将探索如何了解数据库的当前迁移级别，如何从迁移降级，以及如何在某个迁移级别上标记数据库。

13.1 确定数据库的迁移级别

做迁移之前，你应该仔细检查一下，弄清哪些迁移已经应用到数据库中。借助 `alembic current` 命令，你可以知道应用到数据库的最后一次迁移是什么。这个命令会返回当前迁移的修订 ID，并告诉你它是否是最后一次迁移（也称为 head）。让我们在本书示例代码的 CH12/ 文件夹中运行 `alembic current` 命令：

```
# alembic current
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
2e6a6cc63e9 (head) ❶
```

❶ 最后一次应用到数据库的迁移。

一方面，输出指出当前的迁移是 2e6a6cc63e9。另一方面，它也告诉我们这是最后一次迁移。正是在这次迁移中，我们把 `cookies` 表改成了 `new_cookies`。另外，还可以使用 `alembic history` 命令对此进行确认，这个命令会显示一个迁移列表，如示例 13-1 所示。

示例 13-1　迁移历史

```
# alembic history
34044511331 -> 2e6a6cc63e9 (head), Renaming cookies to new_cookies
8a8a9d067 -> 34044511331, Added Cookie model
<base> -> 8a8a9d067, Empty Init
```

示例 13-1 的输出显示了从最开始的空迁移到当前迁移（命名 `cookies` 表）的每个步骤。对于 `new_cookies` 这个表名，我想大家的想法一样，那就是它是一个糟糕的名字，尤其是当我们再次更改 `cookies` 并使用 `new_new_cookies` 这个名字时。为了修正这个问题，将其恢复到以前的名字，我们要进行迁移降级。下面一起学习一下迁移降级。

13.2　迁移降级

要做迁移降级，需要选择想返回的那次迁移的修订 ID。比如，如果你想返回到最初的空迁移，你应该选择的修订 ID 是 8a8a9d067。这里，我们主要想撤销表的重命名。下面使用 `alembic downgrade` 命令恢复到修订 ID 为 34044511331 的状态。

示例 13-2　迁移降级

```
# alembic downgrade 34044511331
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running downgrade 2e6a6cc63e9 -> 34044511331,
      Renaming cookies to new_cookies ❶
```

❶ 降级行。

从示例 13-2 输出中的降级一行可以看出，表名恢复了。我们在第 12 章中学过 `.tables` 命令，这里我们使用它查看表。你会看到如下输出：

```
# sqlite3 alembictest.db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .tables
alembic_version cookies ❶
```

❶ 表名恢复为 `cookies`。

从上面输出的结果可以看到，表名已经恢复为原来的名字（`cookies`）。下面使用 `alembic current` 命令查看数据库所处的迁移级别：

```
± alembic current
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
34044511331 ❶
```

❶ 添加 `cookies` 模型迁移的修订 ID。

可以看到，我们回到了 `downgrade` 命令中指定的迁移。如果想把应用程序恢复到工作状态，还需要更新应用程序代码，以便使用 `cookies` 表名，就像我们在第 12 章中所做的那样。你可能还会发现，我们不在最新的迁移中，因为 `head` 不在最后一行。这个问题需要处理一下。如果不想再把重命名的 `cookies` 用于 `new_cookies` 迁移，只需将其从 alembic/versions/ 目录中删除。如果你置之不理，那么当下一次运行 `alembic upgrade head` 时，它就会运行，这可能会导致灾难性的后果。这里，我们留着它，是为了方便学习如何为数据库标记迁移级别。

13.3　标记数据库迁移级别

当我们要执行诸如跳过迁移或恢复数据库之类的操作时，数据库可能会认为我们的迁移和实际情况不同。为了纠正这个问题，需要明确地把数据库标记成某个特定的迁移级别。在示例 13-3 中，我们将使用 `alembic stamp` 命令把数据库标记为修订 ID2e6a6cc63e9。

示例 13-3　标记数据库迁移级别

```
# alembic stamp 2e6a6cc63e9
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running stamp_revision 34044511331
      -> 2e6a6cc63e9
```

从示例 13-3 可以看到，`alembic stamp` 命令把“revision 34044511331”标记为当前数据库的迁移级别。但是，如果我们查看数据库表，仍然会看到 `cookies` 表。标记数据库迁移级别并不会实际运行迁移，它只是更新 Alembic 表，以此反映我们在命令中提供的迁移级别。这实际上跳过了 34044511331 迁移。



如果你像这样跳过某个迁移，那么它只适用于当前数据库。如果你更改 `sqlalchemy.url`，让 Alembic 迁移环境指向另一个数据库，而新数据库又低于跳过的迁移的级别或者为空，那么你可以运行 `alembic upgrade head` 命令来应用这个迁移。

13.4　生成SQL

有时，你会希望使用 SQL 更改生产数据库的模式，Alembic 对此提供了支持。这在变更管理控制更为严格的环境中很常见，对于那些拥有大规模分布式环境并且需要运行许多不同数据库服务器的环境来说也是如此。这个过程和我们在第 12 章做的“online”Alembic 升级是一样的。我们可以指定 SQL 的起始版本和结束版本。如果你不提供初始迁移，Alembic 将创建用于从空数据库升级的 SQL 脚本。示例 13-4 演示了如何为我们重命名的迁移生成 SQL。

示例 13-4　生成重命名迁移 SQL

```
# alembic upgrade 34044511331:2e6a6cc63e9 --sql ❶
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Generating static SQL
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 34044511331 -> 2e6a6cc63e9,
      Renaming cookies to new_cookies
-- Running upgrade 34044511331 -> 2e6a6cc63e9

ALTER TABLE cookies RENAME TO new_cookies;

UPDATE alembic_version SET version_num='2e6a6cc63e9'
WHERE alembic_version.version_num = '34044511331';
```

❶ 从 34044511331 升级到 2e6a6cc63e9。

示例 13-4 的输出显示了重命名 `cookies` 表所需的两条 SQL 语句，并将 Alembic 迁移级别更新到修订 ID 2e6a6cc63e9 指定的新级别。可以把标准输出重定向到我们选择的文件，以便将其写入文件，如下所示：

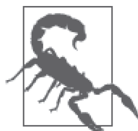
```
# alembic upgrade 34044511331:2e6a6cc63e9 --sql > migration.sql
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Generating static SQL
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 34044511331 -> 2e6a6cc63e9,
      Renaming cookies to new_cookies
```

命令运行完毕后，可以使用 `cat migration.sql` 命令查看 SQL 语句，如下所示：

```
# cat migration.sql
-- Running upgrade 34044511331 -> 2e6a6cc63e9

ALTER TABLE cookies RENAME TO new_cookies;
UPDATE alembic_version SET version_num='2e6a6cc63e9'
WHERE alembic_version.version_num = '34044511331';
```

准备好 SQL 语句之后，就可以使用数据库服务器中的工具来运行它们了。



如果你在一个数据库后端（比如 SQLite）上开发，在另一个数据库（比如 PostgreSQL）上部署，那么一定要修改 `sqlalchemy.url` 配置（我们曾在第 11 章中配置过），将其更改为 PostgreSQL 数据库的连接字符串。如果不这样做，那你得到的 SQL 很可能不适合你的生产数据库。为了避免出现这样的问题，我总是在实际生产中使用的数据库上进行开发。

本章我们讲解了 Alembic 的基础知识，学习了如何查看当前迁移级别、迁移降级和创建 SQL 脚本（这些脚本可以直接应用在生产数据库上，而无须使用 Alembic）。如果想了解更多有关 Alembic 的内容，比如如何处理代码分支，可以查看 Alembic 文档。

下一章会进一步介绍 SQLAlchemy 的用法，包括如何在 Flask Web 框架中使用它。

第 14 章 SQLAlchemy的高级应用

前面章节详细讲解了 SQLAlchemy 的方方面面。与前面不同，本章不会讲得那么细，你可以把本章内容看作对一些有用工具的快速讲解。每一节的末尾都会提供一些参考信息，如果你感兴趣，可以从中学到更多相关知识。请注意，各节内容并不是完整的教程，而是关于如何完成特定任务的简短指南。本章第一部分主要介绍 SQLAlchemy 的一些高级用法，第二部分介绍 SQLAlchemy 如何与 Flask 等 Web 框架结合使用，最后一部分介绍一些可以与 SQLAlchemy 一起使用的库。

14.1 混合属性

混合属性在作为类方法访问时表现出一种行为，而通过实例访问时又表现出另一种行为。我们也可以这样想，当混合属性用在 SQLAlchemy 语句中时会生成有效的 SQL，而通过实例访问时会直接对实例执行 Python 代码。要理解这一点，我发现最好的方法是看代码。下面使用 `Cookie` 类对此进行说明，如示例 14-1 所示。

示例 14-1　`Cookie` 用户数据模型

```
from datetime import datetime

from sqlalchemy import Column, Integer, Numeric, String, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.hybrid import hybrid_property, hybrid_method
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Base = declarative_base()
```

```
class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    @hybrid_property ❶
    def inventory_value(self):
        return self.unit_cost * self.quantity

    @hybrid_method ❷
    def bake_more(self, min_quantity):
        return self.quantity < min_quantity

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
            "cookie_recipe_url='{self.cookie_recipe_url}', " \
            "cookie_sku='{self.cookie_sku}', " \
            "quantity={self.quantity}, " \
            "unit_cost={self.unit_cost})".format(self=self)


Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
```

❶ 创建一个混合属性。

❷ 创建一个混合方法，因为我们需要一个额外的输入来做比较。

在示例 14-1 中，`inventory_value` 是一个混合属性，用来计算 `Cookie` 数据类的两个属性的乘积，而 `bake_more` 是一个混合方法，它需要一个额外的输入来确定我们是否应该烤更多 cookie。（这样有可能返回 false 吗？总是有地方来存放更多 cookie！）数据类定义好之后，就可以开始了解“混合”的真正含义了。让我们看看在查询中使用时 `inventory_value` 是如何工作的（见示例 14-2）。

示例 14-2　混合属性：类

```
print(Cookie.inventory_value < 10.00)
```

示例 14-2 的输出结果如下：

```
cookies.unit_cost * cookies.quantity < :param_1
```

从示例 14-2 的输出中可以看到，混合属性被扩展为一个有效的 SQL 子句。这意味着我们可以对这些属性进行筛选、排序、分组，以及应用数据库函数。下面再打印 `bake_more` 混合方法看看：

```
print(Cookie.bake_more(12))
```

输出结果如下：

```
cookies.quantity < :quantity_1
```

同样，它使用混合方法中的 Python 代码中构建了一个有效的 SQL 子句。为了了解将其作为实例方法访问时会发生什么，我们需要向数据库添加一些数据，如示例 14-3 所示。

示例 14-3　向数据库添加一些记录

```
session = Session()
cc_cookie = Cookie(cookie_name='chocolate chip',
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=12,
                   unit_cost=0.50)
dcc = Cookie(cookie_name='dark chocolate chip',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
             cookie_sku='CC02',
             quantity=1,
             unit_cost=0.75)
mol = Cookie(cookie_name='molasses',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
             cookie_sku='MOL01',
             quantity=1,
             unit_cost=0.80)
session.add(cc_cookie)
session.add(dcc)
session.add(mol)
session.flush()
```

添加了这些数据之后，我们就可以看看通过实例使用混合属性和混合方法时会发生什么。我们可以访问 `dcc` 实例的 `inventory_value` 属性，如下所示：

```
dcc.inventory_value
0.75
```

可以看到，通过实例使用 `inventory_value` 时，它会执行该属性内部的 Python 代码。这正是混合属性或混合方法的魔力所在。还可以通过实例调用 `bake_more` 混合方法，如下所示：

```
dcc.bake_more(12)
True
```

如你所料，当通过实例调用 `bake_more` 混合方法时，它会执行自身的 Python 代码。到这里，我们已经了解了混合属性和混合方法的工作原理，接下来学习一下如何在查询中使用它们。首先使用 `inventory_value` 属性，如示例 14-4 所示。

示例 14-4　在查询中使用混合属性

```
from sqlalchemy import desc

for cookie in session.query(Cookie).order_by(desc(Cookie.inventory_value)):
    print('{:>20} - {:.2f}'.format(cookie.cookie_name, cookie.inventory_value))
```

运行示例 14-4，会得到如下结果：

```
     chocolate chip - 6.00
           molasses - 0.80
dark chocolate chip - 0.75
```

从运行结果可以看出，混合属性的行为和 ORM 的类属性是一样的。在示例 14-5 中，我们将使用 `bake_more` 方法来确定需要烤哪种 cookie。

示例 14-5　在查询中使用混合方法

```
for cookie in session.query(Cookie).filter(Cookie.bake_more(12)):
    print('{:>20} - {}'.format(cookie.cookie_name, cookie.quantity))
```

运行示例 14-5，会得到如下结果：

```
dark chocolate chip - 1
           molasses - 1
```

这和我们希望的一样。虽然这些行为来自于我们的代码，但是混合属性可以做更多事情。你可以阅读 SQLAlchemy 文档中有关混合属性的部分，学习更多相关内容。

14.2　关联代理

关联代理是一个跨关系的指针，它指向某个特定属性。通过关联代理，我们可以轻松地跨关系访问某个属性。例如，当我们想要一份制作 cookie 所需的原料名称列表时，关联代理就能派上用场。下面来看看它是如何处理一段典型的关系的。cookie 和原料之间是多对多的关系。示例 14-6 创建了数据模型及其关系。

示例 14-6　创建模型和关系

```
from datetime import datetime
from sqlalchemy import (Column, Integer, Numeric, String, Table,
                        ForeignKey, create_engine)
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Base = declarative_base()
Session = sessionmaker(bind=engine)
```

```
cookieingredients_table = Table('cookieingredients', Base.metadata, ❶
    Column('cookie_id', Integer, ForeignKey("cookies.cookie_id"),
           primary_key=True),
    Column('ingredient_id', Integer, ForeignKey("ingredients.ingredient_id"),
           primary_key=True)
)


class Ingredient(Base):
    __tablename__ = 'ingredients'

    ingredient_id = Column(Integer, primary_key=True)
    name = Column(String(255), index=True)

    def __repr__(self):
        return "Ingredient(name='{self.name}')".format(self=self)


class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    ingredients = relationship("Ingredient",
                               secondary=cookieingredients_table) ❷

    def __repr__(self):
    return "Cookie(cookie_name='{self.cookie_name}', " \
                   "cookie_recipe_url='{self.cookie_recipe_url}', " \
                   "cookie_sku='{self.cookie_sku}', " \
                   "quantity={self.quantity}, " \
                   "unit_cost={self.unit_cost})".format(self=self)

Base.metadata.create_all(engine)
```

❶ 为多对多关系创建 cookieingredients 表。

❷ 通过 cookieingredients 表创建与原料的关系。

由于这是一个多对多的关系，所以需要创建一个表来映射多个关系。为此，我们创建了 cookieingredients_table。接下来需要添加一种 cookie 和一些原料，如下所示：

```
session = Session()
cc_cookie = Cookie(cookie_name='chocolate chip', ❶
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=12,
                   unit_cost=0.50)
```

```
flour = Ingredient(name='Flour') ❷
sugar = Ingredient(name='Sugar')
egg = Ingredient(name='Egg')
cc = Ingredient(name='Chocolate Chips')
cc_cookie.ingredients.extend([flour, sugar, egg, cc]) ❸
session.add(cc_cookie)
session.flush()
```

❶ 创建 cookie。

❷ 创建原料。

❸ 添加原料至 `cc_cookie.ingredients`。

为了把原料添加到 cookie，我们必须创建原料，然后把它们添加到 `cc_cookie` 的 `ingredients` 属性。现在，如果想列出所有原料的名称（这是我们最初的目标），必须遍历所有原料并获得 `name` 属性。可以使用如下代码来实现这个目标：

```
[ingredient.name for ingredient in cc_cookie.ingredients]
```

返回结果如下：

```
['Flour', 'Sugar', 'Egg', 'Chocolate Chips']
```

如果只想从原料获取 `name` 属性，这么做可能会很麻烦。使用传统关系还需要我们手动创建每种原料，并将其添加到 cookie 中。如果需要确定某种原料是否已经存在，就需要做更多工作。这种情况下，就要用到关联代理了，它可以大大简化这种用法。

可以使用关联代理来访问属性和创建原料。要创建关联代理，需要做如下三件事。

- 导入关联代理。
- 向目标对象添加 `__init__` 方法，方便使用所需值新建实例。
- 创建关联代理，它针对的是你想代理的表名和列名。

我们会启动一个新的 Python shell，并再次创建我们的类。不过，这一次我们要添加一个关联代理，以便更容易地访问原料名称，如示例 14-7 所示。

示例 14-7　创建关联代理

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Session = sessionmaker(bind=engine)

from datetime import datetime

from sqlalchemy import Column, Integer, Numeric, String, Table, ForeignKey
```

```
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.associationproxy import association_proxy ❶


Base = declarative_base()


cookieingredients_table = Table('cookieingredients', Base.metadata,
    Column('cookie_id', Integer, ForeignKey("cookies.cookie_id"),
           primary_key=True),
    Column('ingredient_id', Integer, ForeignKey("ingredients.ingredient_id"),
           primary_key=True)
)


class Ingredient(Base):
    __tablename__ = 'ingredients'
    ingredient_id = Column(Integer, primary_key=True)
    name = Column(String(255), index=True)

    def __init__(self, name): ❷
        self.name = name

    def __repr__(self):
        return "Ingredient(name='{self.name}')".format(self=self)


class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    ingredients = relationship("Ingredient",
                               secondary=cookieingredients_table)

    ingredient_names = association_proxy('ingredients', 'name') ❸


    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
                       "cookie_recipe_url='{self.cookie_recipe_url}', " \
                       "cookie_sku='{self.cookie_sku}', " \
                       "quantity={self.quantity}, " \
                       "unit_cost={self.unit_cost})".format(self=self)


Base.metadata.create_all(engine)
```

❶ 导入关联代理。

❷ 定义 `__init__` 方法，它只需要一个名称。

❸ 创建到原料 `name` 属性的关联代理，并使用 `ingredient_names` 引用它。

上面三件事做完之后，就可以像前面那样使用我们的类了：

```
session = Session()
cc_cookie = Cookie(cookie_name='chocolate chip',
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=12,
                   unit_cost=0.50)
dcc = Cookie(cookie_name='dark chocolate chip',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
             cookie_sku='CC02',
             quantity=1,
             unit_cost=0.75)
flour = Ingredient(name='Flour')
sugar = Ingredient(name='Sugar')
egg = Ingredient(name='Egg')
cc = Ingredient(name='Chocolate Chips')
cc_cookie.ingredients.extend([flour, sugar, egg, cc])
session.add(cc_cookie)
session.add(dcc)
session.flush()
```

这里，关系仍然像我们预想的那样正常工作。不过，为了获得原料名称，可以使用关联代理来避开之前用过的列表推导式。下面是使用 `ingrendient_names` 的例子。

```
cc_cookie.ingredient_names
```

得到的结果如下：

```
['Flour', 'Sugar', 'Egg', 'Chocolate Chips']
```

这比遍历所有原料来创建原料名列表要容易得多。这里，我们忘了添加一种关键原料——油。可以使用关联代理来添加这种新原料，如下所示：

```
cc_cookie.ingredient_names.append('Oil')
session.flush()
```

当我们这样做的时候，关联代理会使用 `Ingredient.__init__` 方法自动为我们创建一种新原料。但需要注意的是，如果我们已经有了一种名为 Oil 的原料，关联代理还是会为我们创建它。这可能会导致出现重复的记录，或者在添加违反约束条件时产生异常。

为了解决这个问题，可以在使用关联代理之前先查询一下现有原料。示例 14-8 演示了具体做法。

示例 14-8　过滤原料

```
dcc_ingredient_list = ['Flour', 'Sugar', 'Egg', 'Dark Chocolate Chips',
                       'Oil'] ❶
existing_ingredients = session.query(Ingredient).filter(
    Ingredient.name.in_(dcc_ingredient_list)).all() ❷
missing = set(dcc_ingredient_list) - set([x.name for x in
                                          existing_ingredients]) ❸
```

❶ 为我们的 dark chocolate chip cookie 定义原料列表。

❷ 查询数据库中已经存在的原料。

❸ 通过两个原料集合的差集找到缺失的原料。

找到已经有的原料之后，将其与所需原料列表进行比较，判断少了什么，这里是“Dark Chocolate Chips”。接下来就可以通过关系添加所有现有原料，而后通过关联代理添加新原料，如示例 14-9 所示。

示例 14-9　向 dark chocolate chip cookie 添加原料

```
dcc.ingredients.extend(existing_ingredients) ❶
dcc.ingredient_names.extend(missing) ❷
```

❶ 通过关系添加现有原料。

❷ 通过关联代理添加新原料。

到这里，我们就可以使用如下代码把原料名打印出来了：

```
dcc.ingredient_names
```

这会得到如下结果：

```
['Egg', 'Flour', 'Oil', 'Sugar', 'Dark Chocolate Chips']
```

当我们把新旧原料添加到 cookie 时，这使得我们能够快速处理它们，并且所得到的输出正是我们想要的。关联代理还有许多其他用途，你可以在关联代理文档中了解更多信息。

14.3　集成SQLAlchemy和Flask

我们经常看到 SQLAlchemy 与 Flask Web 应用程序一起使用。Flask 的创建者创建了一个 Flask-SQLalchemy 包来简化集成过程。使用 Flask-SQLalchemy 会得到预配置作用域会话，它们与 Flask 应用程序的页面生命周期绑定在一起。可以使用 pip 来安装 Flask-SQLalchemy，如下所示：

```
# pip install flask-sqlalchemy
```

在使用 Flask-SQLalchemy 时，强烈建议你使用 app 工厂模式，这不是 Flask-SQLalchemy

文档快速上手部分使用的模式。app 工厂模式会使用一个函数把所有合适的附加组件和配置组装成一个应用程序。这通常放在应用程序的 app/__init__.py 文件中。示例 14-10 演示了如何构造 `create_app` 方法。

示例 14-10　app 工厂函数

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

from config import config

db = SQLAlchemy() ❶


def create_app(config_name):❷
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    db.init_app(app) ❸
    return app
```

❶ 创建一个未配置的 Flask-SQLAlchemy 实例。

❷ 定义 `create_app` 工厂。

❸ 使用 `app` 上下文初始化实例。

app 工厂需要一个配置文件来定义 Flask 设置和 SQLAlchemy 连接字符串。在示例 14-11 中，我们在应用程序根目录下定义了一个配置文件 config.py。

示例 14-11　Flask app 配置

```
import os


basedir = os.path.abspath(os.path.dirname(__file__))


class Config: ❶
    SECRET_KEY = 'development key'
    ADMINS = frozenset(['jason@jasonamyers.com', ])


class DevelopmentConfig(Config): ❷
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = "sqlite:////tmp/dev.db"


class ProductionConfig(Config): ❸
    SECRET_KEY = 'Prod key'
    SQLALCHEMY_DATABASE_URI = "sqlite:////tmp/prod.db"


config = { ❹
```

```
        'development': DevelopmentConfig,
        'production': ProductionConfig,
        'default': DevelopmentConfig
    }
```

❶ 定义基本 Falsk 配置。

❷ 定义开发时使用的配置。

❸ 定义实际生产环境中使用的配置。

❹ 定义一个字典，用来完成从简单名称到配置类的映射。

准备好 app 工厂和配置之后，我们就有了一个功能正常的 Flask 应用程序。接下来，可以定义数据类了。在 apps/models.py 中，定义 `Cookie` 模型，如示例 14-12 所示。

示例 14-12 定义 `Cookie` 模型

```
from app import db


class Cookie(db.Model):
    __tablename__ = 'cookies'

    cookie_id = db.Column(db.Integer(), primary_key=True)
    cookie_name = db.Column(db.String(50), index=True)
    cookie_recipe_url = db.Column(db.String(255))
    quantity = db.Column(db.Integer())
```

在示例 14-12 中，我们使用了在 app/__init__.py 中创建的 `db` 实例来访问 SQLAlchemy 库的大部分。现在，可以在查询中使用 `Cookie` 模型了，就像我们讲 ORM 时所做的那样。唯一的区别是，现在我们的会话嵌套在 `db.session` 对象中。

此外，Flask-SQLAlchemy 在每个 ORM 数据类中添加了 `query` 方法，这在普通 SQLAlchemy 代码中是看不到的。强烈建议你不要使用这种语法，因为这在与其他 SQLAlchemy 查询混合配用时可能会引起混乱。这种查询风格看起来像下面这样：

```
Cookie.query.all()
```

学完上面这些内容，你应该了解了如何将 SQLAlchemy 与 Flask 集成在一起。你可以阅读 Flask-SQLalchemy 文档了解更多内容。Miguel Grinberg 写了一本很棒的书——《Flask Web 开发》[1]，其中介绍了如何构建一个完整的应用程序。

14.4 SQLAcodegen

我们在第 5 章学习了 SQLAlchemy Core 的反射，并在第 10 章学习了与 ORM 结合使用的自动映射（automap）。虽然这两种解决方案都很棒，但是它们要求每次重启应用程序时

注 1：此书已由人民邮电出版社出版，详情请见 http://www.ituring.com.cn/book/2463。——编者注

都执行反射，或者在动态模块的情况下，每次重新加载模块时都执行反射。SQLAcodegen 使用反射构建一系列 ORM 数据类，你可以在应用程序代码库中使用它们来避免多次反射数据库。SQLAcodegen 能够检测出多对一、一对一和多对多的关系。可以通过 pip 安装 SQLAcodegen，如下所示：

```
pip install sqlacodegen
```

要运行 SQLAcodegen，需要为它指定一个数据库连接字符串来进行连接。我们使用前面用过的 Chinook 数据库的副本（你可以在本书示例代码的 CH14/ 文件夹中找到它）。在 CH14/ 文件夹中，使用合适的连接字符串运行 SQLAcodegen 命令，如示例 14-13 所示。

示例 14-13　针对 Chinook 数据库运行 SQLAcodegen

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite ❶

# coding: utf-8
from sqlalchemy import (Table, Column, Integer, Numeric, Unicode, DateTime,
                        ForeignKey)
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base


Base = declarative_base()
metadata = Base.metadata


class Album(Base):
    __tablename__ = 'Album'

    AlbumId = Column(Integer, primary_key=True, unique=True)
    Title = Column(Unicode(160), nullable=False)
    ArtistId = Column(ForeignKey(u'Artist.ArtistId'), nullable=False, index=True)

    Artist = relationship(u'Artist')

class Artist(Base):
    __tablename__ = 'Artist'

    ArtistId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))

class Customer(Base):
    __tablename__ = 'Customer'

    CustomerId = Column(Integer, primary_key=True, unique=True)
    FirstName = Column(Unicode(40), nullable=False)
    LastName = Column(Unicode(20), nullable=False)
    Company = Column(Unicode(80))
    Address = Column(Unicode(70))
    City = Column(Unicode(40))
    State = Column(Unicode(40))
    Country = Column(Unicode(40))
```

```
    PostalCode = Column(Unicode(10))
    Phone = Column(Unicode(24))
    Fax = Column(Unicode(24))
    Email = Column(Unicode(60), nullable=False)
    SupportRepId = Column(ForeignKey(u'Employee.EmployeeId'), index=True)

    Employee = relationship(u'Employee')

class Employee(Base):
    __tablename__ = 'Employee'

    EmployeeId = Column(Integer, primary_key=True, unique=True)
    LastName = Column(Unicode(20), nullable=False)
    FirstName = Column(Unicode(20), nullable=False)
    Title = Column(Unicode(30))
    ReportsTo = Column(ForeignKey(u'Employee.EmployeeId'), index=True)
    BirthDate = Column(DateTime)
    HireDate = Column(DateTime)
    Address = Column(Unicode(70))
    City = Column(Unicode(40))
    State = Column(Unicode(40))
    Country = Column(Unicode(40))
    PostalCode = Column(Unicode(10))
    Phone = Column(Unicode(24))
    Fax = Column(Unicode(24))
    Email = Column(Unicode(60))

    parent = relationship(u'Employee', remote_side=[EmployeeId])

class Genre(Base):
    __tablename__ = 'Genre'

    GenreId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Invoice(Base):
    __tablename__ = 'Invoice'

    InvoiceId = Column(Integer, primary_key=True, unique=True)
    CustomerId = Column(ForeignKey(u'Customer.CustomerId'), nullable=False,
                        index=True)
    InvoiceDate = Column(DateTime, nullable=False)
    BillingAddress = Column(Unicode(70))
    BillingCity = Column(Unicode(40))
    BillingState = Column(Unicode(40))
    BillingCountry = Column(Unicode(40))
    BillingPostalCode = Column(Unicode(10))
    Total = Column(Numeric(10, 2), nullable=False)

    Customer = relationship(u'Customer')


class InvoiceLine(Base):
    __tablename__ = 'InvoiceLine'
```

```
    InvoiceLineId = Column(Integer, primary_key=True, unique=True)
    InvoiceId = Column(ForeignKey(u'Invoice.InvoiceId'), nullable=False, index=True)
    TrackId = Column(ForeignKey(u'Track.TrackId'), nullable=False, index=True)
    UnitPrice = Column(Numeric(10, 2), nullable=False)
    Quantity = Column(Integer, nullable=False)

    Invoice = relationship(u'Invoice')
    Track = relationship(u'Track')


class MediaType(Base):
    __tablename__ = 'MediaType'

    MediaTypeId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))

class Playlist(Base):
    __tablename__ = 'Playlist'

    PlaylistId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))

    Track = relationship(u'Track', secondary='PlaylistTrack')

t_PlaylistTrack = Table(
    'PlaylistTrack', metadata,
    Column('PlaylistId', ForeignKey(u'Playlist.PlaylistId'), primary_key=True,
           nullable=False),
    Column('TrackId', ForeignKey(u'Track.TrackId'), primary_key=True, nullable=False,
           index=True),
    Index('IPK_PlaylistTrack', 'PlaylistId', 'TrackId', unique=True)
)


class Track(Base):
    __tablename__ = 'Track'

    TrackId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(200), nullable=False)
    AlbumId = Column(ForeignKey(u'Album.AlbumId'), index=True)
    MediaTypeId = Column(ForeignKey(u'MediaType.MediaTypeId'), nullable=False,
                         index=True)
    GenreId = Column(ForeignKey(u'Genre.GenreId'), index=True)
    Composer = Column(Unicode(220))
    Milliseconds = Column(Integer, nullable=False)
    Bytes = Column(Integer)
    UnitPrice = Column(Numeric(10, 2), nullable=False)

    Album = relationship(u'Album')
    Genre = relationship(u'Genre')
    MediaType = relationship(u'MediaType')
```

❶ 对本地的 Chinook SQLite 数据库运行 SQLAcodegen。

如示例 14-13 所示，运行这个命令时，它会创建一个完整的文件，其中包含数据库的所有 ORM 数据类以及正确的导入。这个文件可以在我们的应用程序中使用。如果 `Base` 对象是在其他地方创建的，你可能需要调整它的设置。还可以通过使用 `--tables` 参数仅为几个表生成代码。在示例 14-14 中，我们选择对 `Artist` 和 `Track` 表运行 SQLAcodegen。

示例 14-14　对 `Artist` 和 `Track` 表运行 SQLAcodegen

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track ❶

# coding: utf-8
from sqlalchemy import Column, ForeignKey, Integer, Numeric, Unicode
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base


Base = declarative_base()
metadata = Base.metadata


class Album(Base):
    __tablename__ = 'Album'

    AlbumId = Column(Integer, primary_key=True, unique=True)
    Title = Column(Unicode(160), nullable=False)
    ArtistId = Column(ForeignKey(u'Artist.ArtistId'), nullable=False, index=True)

    Artist = relationship(u'Artist')

class Artist(Base):
    __tablename__ = 'Artist'

    ArtistId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Genre(Base):
    __tablename__ = 'Genre'

    GenreId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class MediaType(Base):
    __tablename__ = 'MediaType'

    MediaTypeId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Track(Base):
    __tablename__ = 'Track'
    TrackId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(200), nullable=False)
```

```
        AlbumId = Column(ForeignKey(u'Album.AlbumId'), index=True)
        MediaTypeId = Column(ForeignKey(u'MediaType.MediaTypeId'), nullable=False,
                             index=True)
        GenreId = Column(ForeignKey(u'Genre.GenreId'), index=True)
        Composer = Column(Unicode(220))
        Milliseconds = Column(Integer, nullable=False)
        Bytes = Column(Integer)
        UnitPrice = Column(Numeric(10, 2), nullable=False)

        Album = relationship(u'Album')
        Genre = relationship(u'Genre')
        MediaType = relationship(u'MediaType')
```

❶ 对本地 Chinook SQLite 数据库运行 SQLAcodegen，但只针对 `Artist` 和 `Track` 两个表。

当我们在示例 14-14 中指定 `Artist` 和 `Track` 两个表时，SQLAcodegen 会为这两个表以及所有与它们有关系的表创建类。SQLAcodegen 这样做是为了确保其构建的代码可以使用。如果想把生成的类直接保存到文件中，可以使用标准重定向，如下所示：

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track > db.py
```

如果想了解 SQLAcodegen 的更多功能，可以使用 `SQLAcodegen  -help` 命令查看更多选项。撰写本书时，尚不存在任何有关 SQLAcodegen 的官方文档。

到这里，本章内容就全部讲完了。希望你喜欢这几个关于 SQLAlchemy 高级用法的例子。

第 15 章

接下来做什么

前面我们学习了如何使用 SQLAlchemy Core 和 SQLAlchemy ORM 以及 Alembic 数据库迁移工具，希望你在这个过程中学得开心。请记住，SQLAlchemy 网站上有大量的文档以及各种会议的演示文稿，通过它们，你可以更详细地了解特定主题。PyVideo 网站上也有一些关于 SQLAlchemy 的演讲视频，其中有几个还是我制作的。

下一步怎么做，主要取决于你想用 SQLAlchemy 做什么。

- 如果你想了解关于 Flask 和 SQLAlchemy 的更多信息，请阅读 Miguel Grinberg 写的《Flask Web 开发》一书。
- 如果你想学习更多关于测试或如何使用 pytest 的内容，Alex Grönholm 写了几篇优秀的关于有效测试策略的博客文章：第一部分（http://alextechrants.blogspot.fi/2013/08/unit-testing-sqlalchemy-apps.html）和第二部分（http://alextechrants.blogspot.fi/2014/01/unit-testing-sqlalchemy-apps-part-2.html），很值得读一读。
- 如果你想了解更多有关 SQLAlchemy 插件和扩展的内容，请查看 Hong Minhee 的 Awesome SQLAlchemy。其中介绍了大量与 SQLAlchemy 相关的技术。
- 如果你想了解更多有关 SQLAlchemy 内部原理的内容，可以阅读 Mike Bayer 撰写的“The Architecture of Open Source Applications”一文。

希望你在这本书中学到的知识能够帮你打下坚实的基础，助你进一步提高自己的技能。祝愿你在今后的工作、学习和生活中一切顺利！

关于作者

贾森 • 迈尔斯（Jason Myers）在思科担任 OpenStack 软件工程师。几年前他转做开发，在那之前，他做过几年系统架构师，为几家大型科技公司、医院、体育场和电信公司构建数据中心和云架构。他是一位富有激情的开发者，经常在一些技术活动中发表演讲。他还是 PyTennessee 会议的主席。他喜欢解决与健康有关的问题，并主持 Sucratrend 项目，致力于帮助糖尿病患者管理病情，提高他们的生活质量。他曾在 Web、数据仓库和分析应用程序中使用 SQLAlchemy 库。

里克 • 科普兰（Rick Copeland）是 Synapp.io 的联合创始人兼首席执行官，Synapp.io 是一家总部位于亚特兰大的公司，专为从事电子邮件营销的企业提供 SaaS 解决方案。他也是一位经验丰富的 Python 开发人员，专注研究关系型数据库和 NoSQL 数据库，并因对 MongoDB 社区的贡献而被 MongoDB 公司授予"MongoDB 大师"称号。他经常在各种用户组和会议上发表演讲，也是亚特兰大创业社区的积极分子。

关于封面

本书封面上的动物是一种大型飞鱼（Cypselurus oligolepis，少鳞燕鳐鱼）。飞鱼是飞鱼科（Exocoetidae）的通俗叫法，该科由大约 40 个种属组成，主要生活在大西洋、太平洋和印度洋的温暖热带和亚热带水域。飞鱼体长 18 ～ 30 厘米，它们的特点是长着巨大的像翅膀一样的胸鳍。有些种属也长有很大的腹鳍，因此也被称为四翼飞鱼。

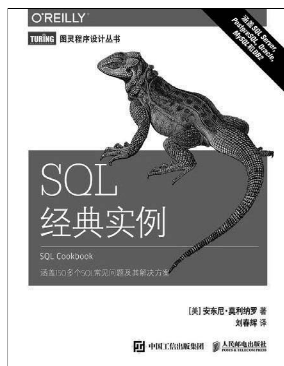
顾名思义，飞鱼拥有一种独特的能力，那就是跃出水面，在空中滑翔，滑翔距离最长可达四分之一英里（约 400 米）。它们有着鱼雷一样的身体，可以帮助它们以一定的速度从海水中飞出（大约每小时 60 千米），它们长着独特的胸鳍和分叉的尾鳍，这使得它们能够在空中滑翔。生物学家认为，飞鱼这种非凡的本领可能是为了躲避捕食者而进化出来的，这些捕食者包括金枪鱼、鲭鱼、旗鱼、马林鱼和其他大型鱼类。不过，飞鱼有时很难躲避人类捕食者。夜晚，渔民们只要在独木舟上挂上一盏灯，就能吸引飞鱼的注意，它们会纷纷跳进小舟，无法再跳回水中了。

飞鱼干是居住在中国台湾兰屿的雅美人（自称达悟人）的主食，而飞鱼籽在日本菜中很常见。在"飞鱼之乡"巴巴多斯，飞鱼也是令人垂涎的美味佳肴，然而如今海水污染和过度捕捞已导致飞鱼数量锐减。不过，飞鱼在那里仍然有显著的文化地位，它是国菜（库库和飞鱼，coucou and flyling fish）的主要原料，在硬币、艺术品甚至巴巴多斯旅游局（Barbados Tourism Authority）的标识上都能见到飞鱼的身影。

O'Reilly 图书封面上的许多动物都濒临灭绝，这些动物对这个世界很重要。要想了解更多关于如何提供帮助的信息，请访问 animals.oreilly.com。

封面图片来自 *Dover's Animals*。

技术改变世界 · 阅读塑造人生

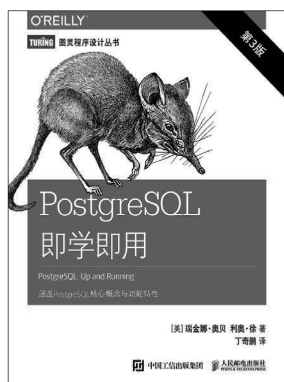


SQL 经典实例

◆ **涵盖150多个SQL常见问题及其解决方案**

作者： 安东尼 • 莫利纳罗

译者： 刘春辉

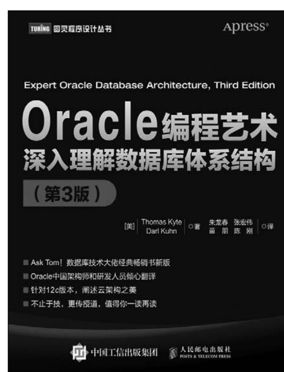


PostgreSQL 即学即用（第 3 版）

◆ **涵盖PostgreSQL核心概念与功能特性**

作者： 瑞金娜 • 奥贝 利奥 • 徐

译者： 丁奇鹏



Oracle 编程艺术：深入理解数据库体系结构（第 3 版）

◆ **Ask Tom！Oracle数据库技术大佬经典畅销书新版**

◆ **针对12c版本，阐述云架构之美**

作者： Thomas Kyte　Darl Kuhn

译者： 朱龙春　张宏伟　苗朋　陈刚



微信连接



回复“数据库”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

SQLAlchemy: Python数据库实战

SQLAlchemy是一个流行的开源代码库，功能强大又相当灵活，能够帮助Python程序员使用各种关系型数据库，许多公司甚至把SQLAlchemy看作在Python中使用关系型数据库的标准方式。本书通过真实示例，演示了如何使用SQLAlchemy构建简单的数据库应用程序，以及如何使用相同的元数据同时连接多个数据库。

如果你是一位中级Python开发人员，掌握了基本的SQL语法和关系理论知识，那么对你而言本书既是很棒的学习工具，也是不错的参考手册。

- **SQLAlchemy Core：借助SQL表达式语言以Python方式向应用程序提供数据库服务。**
- **SQLAlchemy ORM：使用对象关系映射器将数据库模式和操作绑定到应用程序中的数据对象上。**
- **Alembic：随着应用程序的演进，灵活地处理需要对数据库做的更改。**
- **高级应用：将SQLAlchemy与Flask Web框架及SQLAcodegen库结合使用。**

“本书选用了大量清晰、简洁的示例，非常适合使用关系型数据库的Python开发者阅读。从简单查询到高级ORM访问再到模式迁移，本书几乎涵盖了方方面面，足以帮助读者将应用程序连接到任何一种关系型数据库。”

——**Doug Hellman**

惠普公司OpenStack贡献者，*The Python Standard Library by Example*作者

贾森·迈尔斯（Jason Myers），Built Technologies平台首席工程师，Juice Analytics公司高级开发者，曾在思科公司担任技术主管。在转做开发前，曾做过15年系统架构师。

里克·科普兰（Rick Copeland），Carefolio公司联合创始人兼CEO，Arborian咨询公司首席顾问，是位经验丰富的创业者、技术主管、演讲者、培训师和顾问。

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议 计算机／数据库

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)





ISBN 978-7-115-51630-5
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks